STUDY OF SYSTEM LEVEL MILLIBOTTLENECKS AND THEIR IMPACT ON N-TIER SYSTEM PERFORMANCE

A Thesis
Presented to
The Academic Faculty

by

Tao Zhu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2017

STUDY OF SYSTEM LEVEL MILLIBOTTLENECKS AND THEIR IMPACT ON N-TIER SYSTEM PERFORMANCE

Approved by:

Professor Dr. Calton Pu, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Dr. Shamkant B. Navathe
School of Computer Science
Georgia Institute of Technology

Professor Dr. Ling Liu
School of Computer Science
Georgia Institute of Technology

Professor Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Professor Dr. Qingyang Wang
School of Electrical Engineering and
Computer Science
Louisiana State University

Date Approved: August 2017

To my parents and my wife

ACKNOWLEDGEMENTS

I am extremely grateful to my advisor, Dr. Calton Pu , who guided me tirelessly throughout my PhD. He is a brilliant researcher who is always able to push ideas one level further, provide the input needed to finish a task, and share their experience about the research process. I was especially fortunate to work with him and benefit from his advises. Calton is also like a father, he not only taught me how to become a better researcher, but also helped me to become a better person.

I would also like to give special thanks to the members of my dissertation committee - Dr. Ling Liu, Dr. Shamkant Navathe, Dr. Ada Gavrilovska, and Dr. Qingyang Wang - for serving on my dissertation committee. I very much appreciate the time and effort they took to supervise my dissertation work.

The work in this dissertation is the result of collaboration with my previous and current colleagues in ELBA project. Qingyang Wang, Jack Li, Junhee Park, Chien-An Lai, Josh Kimball, Deepal Jayasinghe, and Aibek Musaev you have supported me all along the way. It has been a privilege to work with such amazing colleagues. Beyond direct collaborators in ELBA project, many fellow graduate student contributed to my graduate work. Qi Zhang, Jian Huang, Yanwei Zhang provided fantastic advice and ideas on my research.

Last but not least, I want to thank my parents Jianzhong and Meijuan and my wife Xiumei for their unconditional support, patience, and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The performance of n-tier web-facing applications often suffer from response time long-tail problem. One of the causes of long-tail problem consists of millibottlenecks that appear and disappear within tens to hundreds of milliseconds. We propose a novel approach to detect system level millibottlenecks by fine-grained monitoring of locks. Through the comprehensive analysis of Linux kernel call graph, we found instrumenting around locks can achieve high coverage and minimize the number of instrumenting points. In this dissertation, we present two case studies in diagnosing the root cause of system level millibottlenecks and their impact on N-tier systems. For the first case study, we use concrete experimental evidence that shows our approach can diagnose the root cause of system level millibottleneck, which is caused by conservative stable page. The millibottleneck is somewhat similar to the priority inversion problem in scheduling, the fundamental cause is CFQ scheduler ignores the priories of asynchronous writes. For the second case study, we have found load balancing policies and mechanisms that appeared to work well in stable environments have exhibited several limitations when facing millibottlenecks. Experiments with standard n-tier benchmarks show that during millibottlenecks, some load balancing policy/mechanism combinations make the mistake of sending new requests to the node(s) suffering from millibottlenecks, instead of the idle nodes as load balancers are supposed to do. Several of these mistakes are due to the implicit assumptions made by load balancing policies and mechanisms on the stability of system state. Our study shows that appropriate remedies at policy and mechanism levels can avoid these mistakes during millibottlenecks and remove the VLRT requests, thus improving the average response time by a factor of 12.

CHAPTER I

INTRODUCTION

The interactive nature of modern web-facing applications requires low and predictable latencies, however, the very long response time(VLRT) requests are a serious and perplexing problem because deviation from strict QoS is bad for business: A study by Amazon [34] reported that every increase in response time of 100 milliseconds is correlated to roughly 1% loss in sales. Similarly, Google found that a 500ms additional delay in returning search results could reduce revenues by up to 20% [36]. Given the number of customers they serve, companies such as Amazon and Google want to reduce the response time long tails to the 99th and 99.9th percentiles [29,30]. The VLRT requests are also a perplexing problem since they appear with modest utilization levels, e.g., 50% of CPU utilization.

One of the causes of VLRT requests consists of millibottlenecks [51–53] that appear and disappear within tens to hundreds of milliseconds. Past studies have examined and delivered valuable insights of causes on the VLRT requests. VLRT requests can have very different causes, traversing the system stack. These include CPU dynamic voltage and frequency scaling (DVFS) control at the architecture layer, Java garbage collection (GC) at the system software layer, virtual machine (VM) consolidation at the VM layer and bursty workloads [51–53]. The millibottlenecks in our experiments are caused by flushing dirty pages.

Millibottleneck can appear at all levels of the software stack, the major focus of this proposal is system level millibottlenecks. Diagnosing the root cause of millibottlenecks at system level is very challenging. First, The kernel can be tightly coupled with hardware or workloads, as such, the millibottlenecks might be very different from one machine to another. For example, the millibottlenecks caused by stable page appear on the machine with disk requires data integrity check, but they will not happen on the machine doesn't require data integrity check. Second, the kernel is a complicated system which consists of more than 10

millions of lines of code. Take IO stack as an example, it consists of the virtual file system (VFS), page cache, Ext4 file system, JBD2 journaling layer, I/O scheduler, and block layer. Revealing the root cause of millibottlenecks may require correlating to the source code. It is very difficult to pinpoint the causes of millibottlenecks. Lastly, millibottlenecks are very short, usually take tens to hundreds of milliseconds, they are often invisible to classic performance monitoring tools. From the Sampling Theorem, millibottlenecks would not be detectable by monitoring tools with sampling periods of multiple seconds or minutes, which is the typical setting of many past and current performance monitoring tools [45].

The system level millibottleneck we have found is caused by conservative stable page, a mechanism doesn't allow a high priority process to modify a dirty page being flushed to disk by the writeback process with lower priority, regardless whether the device performs integrity checking or not. This somewhat similar to the priority inversion problem in scheduling, where a high priority task cannot make any progress as a low priority task has a resource it requires. We envision a new class of system level millibottlenecks caused by priority inversion exists in Linux kernel, especially in I/O stack. The fundamental cause of this class of millibottlenecks is that CFQ scheduler ignores the priorities of asynchronous writes.

Millibottlenecks can propagate through a computer system and have their effects amplified due to dependencies among the components, causing significant performance bugs (variability) [45]. N-tier systems are one the best examples of scalable distributed systems that provide QoS (quality of service, e.g., guaranteed response time) for production web-facing applications such as e-commerce. The scalability of n-tier systems is largely due to: (1) their ability to allocate servers dynamically according to load, and (2) effective load balancers to spread requests among the available servers in each tier to minimize response time and maximize throughput. For web-facing applications, n-tier systems have been able to provide good response times, as fast as milliseconds. However, they are also known for the lingering problem of long-tail distribution of response times, where some very long response time (VLRT) requests take several seconds to return results.

1.1 Dissertation Statement and Contributions

My dissertation statement is formulated as follows:

Thesis Statement: *Detecting and understanding system level millibottlenecks need fine-grained resource monitoring and the impact of millibottlenecks can be amplified by inappropriate resource scheduling policy and mechanism)*

To support my dissertation statement, we make the following concrete contributions:

- **Our first contribution is a novel approach to detect system level millibottlenecks through fine-grained monitoring of locks.** Firstly, we explain why monitoring locks can help diagnosing the millibottlenecks: (1) monitoring locks can achieve more fine-grained than hardware or software resource monitoring; (2) Through the comprehensive analysis of Linux kernel call graph, we found monitoring locks can achieve high coverage of Linux kernel and minimize the number of instrumenting points. High coverage gives us the confidence that our approach can discover the majority of system level millibottlenecks. To achieve this goal, we extend lockstat, a utility gathers and displays kernel locking and profiling statistics, to support fine-grained timescale granularity. Secondly, we introduce our three-stage approach to detect system level millibottlenecks. In step one, we run RUBBoS benchmark with the MilliMonitor and our lock monitoring tool to find when and where the millibottlenecks happen. In step two, we search the contended locks to identify the candidates that cause the VLRT requests. In step three, we correlate the contended lock to the source code to reveal the root causes of system level millibottlenecks. Finally, we evaluate the overhead of our lock monitoring tool and propose several solutions to reduce the overhead.

- **Our second contribution is that we conduct an illustrative scenario in which our approach successfully diagnoses the root cause of millibottlenecks caused by conservative stable page**. We compare our approach with

against perf to demonstrate benefits of our approach to diagnose system level millibottlenecks. By using our approach, we are able to correlate the millibottlenecks with kernel source code. A careful analysis reveals that millibottleneck arises because of conservative stable page where the file system operations are blocked until one or more asynchronous I/O operations are completed. This situation is somewhat similar to the priority inversion problem in scheduling where a high priority task cannot make any progress as a low priority task has a resource it requires. The fundamental cause of this class of millibottlenecks is that CFQ scheduler ignores the priorities of asynchronous writes: (1) It ignores CPU priorities of the process issuing asynchronous writes: I/O priority is not derived from the CPU priority, which means high CPU priority task may not get more I/O resource than low CPU priority task. (2) It ignores I/O priorities of the process issuing asynchronous writes: writeback process (priority-4) submits all the writes on behalf of these processes, no matter what I/O priority the process issuing write request is. Due to the limitation of CFQ scheduler, we believe there is a new class of system level millibottlenecks caused by priority inversion in Linux kernel, especially in I/O stack. We provide a remedy called optimistic stable write in the situation where hardware data integrity features are not in use. The core idea is that it checks if the given page is related to a backing device that requires page contents to be held stable during writeback. If so, then it will wait for any pending writeback to complete. Given that our hardware does not perform data integrity check, experimental results show our remedy significantly improve average response time by 4X.

- **Our third contribution is that we have found load balancing policies and mechanisms that appeared to work well in stable environments have exhibited several limitations when facing millibottlenecks.** The scalability of n-tier systems relies on effective load balancing to distribute load among the servers of the same tier. For example, load balancing policies that use cumulative requests served may send requests to a server that did less work historically, but that is suffering from (sudden) millibottlenecks at this moment. Our experimental evidence

showing several such limitations of current load balancing policies and mechanisms in the presence of millibottlenecks. Several of these mistakes are due to the implicit assumptions made by load balancing policies and mechanisms on the stability of system state. We propose changes to load balancing policies and mechanisms that take into account the millibottlenecks, e.g., adding the consideration of recent utilization changes. Our study shows that appropriate remedies at policy and mechanism levels can avoid these mistakes during millibottlenecks and remove the VLRT requests, thus improving the average response time by a factor of 12. Our remedies can reduce or bypass the emergence of VLRT requests caused by millibottleneck, regardless of the origin of millibottlenecks

1.2 Organization of This Dissertation

The rest of this dissertation is organized as follows. Chapter 2 gives a brief introduction to our approach to discovery and study millibottlenecks. Chapter 3 describes our fine-grained lock monitoring tool and the three-stage approach to detect system level millibottlenecks by using this tool. Chapter 4 uses a concrete example to demonstrate the effectiveness of our approach: we found a system level millibottleneck caused by conservative stable page where the file system operations are blocked until one or more asynchronous I/O operations are completed. This situation is somewhat similar to the priority inversion problem in scheduling where a high priority task cannot make any progress as a low priority task has a resource it requires. Chapter 5 shows the impact of millibottlenecks in N-tier systems. We demonstrate load balancing policies and mechanisms that appeared to work well in stable environments have exhibited several limitations when facing millibottlenecks. Experiments with standard n-tier benchmarks show that during millibottlenecks, some load balancing policy/mechanism combinations make the mistake of sending new requests to the node(s) suffering from millibottlenecks, instead of the idle nodes as load balancers are supposed to do. Several of these mistakes are due to the implicit assumptions made by load balancing policies and mechanisms on the stability of system state. Chapter 6 summarizes the related work and Chapter 7 concludes the dissertation and discusses future work.

CHAPTER II

AN EXPERIMENTAL STUDY OF MILLIBOTTLENECKS

In the chapter, we describe our experimental approach to discovery and study millibottlenecks. We first introduce the RUBBoS benchmark, which monitors the end-to-end response time for each request. We then show the details of the experimental setup. Lastly, we give a brief introduction of MilliMonitor, which will provide fine-grain resource monitoring, low overhead event monitoring, and data analytic tools to diagnose millibottlenecks.

2.1 RUBBoS benchmark

N-tier is a classic architecture of modern web application, which is usually implemented as three or more tiers. RUBBoS benchmark [8] has been widely used in academia research due to its real production system significance. RUBBoS can be configured as a three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC) system. The workload generator emulates real users interacting with the web application. To mimic real user behavior, each client has a thinking time between receiving a web page and submitting a new page download request. The think time is distributed in such a way that the average think time is 7 seconds. The workload includes 24 different interactions such as "register user" or "view story". The benchmark includes two kinds of workload modes: browse-only and read/write interaction mixes. The lengths of story and comment bodies vary from 1KB to 8KB. We use browse-only workload for our evaluation. Our experiments run the standard 3-tier configuration, which uses Apache HTTP server as web server, Tomcat as application server and MySQL as database server.

2.2 Experimental Setup

We ran the experiments on Emulab [4], a network testbed offering researchers a wide range of environments in which to develop, debug, and evaluate their systems. In 3 and 4, we used 7 d710 nodes to run the RUBBoS benchmark, including 1 control node, 1 client nodes,

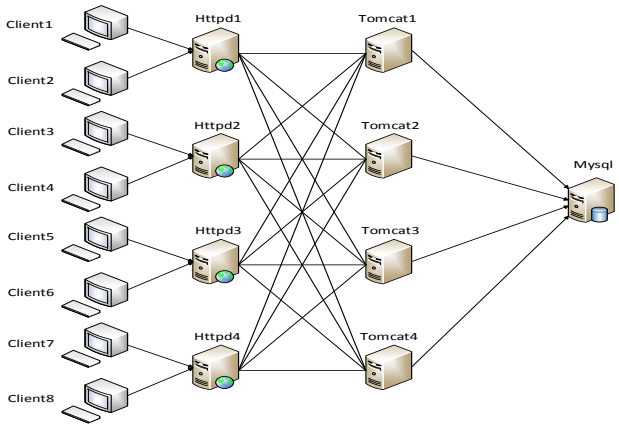


Figure 1: Topology

Table 1: Software Stack & Hardware Configuration

Web Server	Apache HTTPD 2.2.22
Application Server	Apache Tomcat 5.5.17
Database Server	Mysql 5.5.17
Java	jdk 7
Tomcat Connectors	mod_jk 1.2.32
Operating System	Fedora 15(Kernel 3.3)

CPU	Intel Xeon E5530, 2.40GHz Quad-Core
Memory	12 GB
HDD	WD SATA 7,200 RPM, 500GB
Network Link	1Gbps

1 web server, 1 application servers and 1 database server. In 5, we used 18 d710 nodes to run the RUBBoS benchmark, including 1 control node, 8 client nodes, 4 web servers, 4 application servers and 1 database server, the topology is shown in Figure 1. The first two client nodes send requests to the first web server, the third and fourth client node send requests to the second web server, etc. Each web server communicates with 4 application servers via mod_jk [10]. Table 1 summarizes the software stack and hardware configuration we used throughout our evaluation. The configurations for each tier can be found in Table 2.

Table 2: Configurations of Major Software Resources

Tier	Parameter name	Value
Apache	MaxClients	200
	ThreadsPerChild	100
	WorkerConnectionPoolSize	25
Tomcat	maxThreads	210
DB connections	Total	48
	each Servlet	6
Mysql	Query Cache Size	10MB

2.3 Millimonitor

To discovery and study of millibottlenecks, we are building a software toolkit called MilliMonitor [45]. MilliMonitor has multiple product versions and each version has its own set of releases. In this thesis, we mainly focus on alpha version, which consists of three main components.

- Low-overhead generic resource utilization monitors such as sar [17] and collectl [12]. Collectl can run at a very fine-grained monitoring level while incurring very little CPU overhead. We have built parsers for these monitors to extract resource usage information.

- An open source event monitor and logger called Milliscope [38]. Milliscope adopts techniques such as software specialization and AOP(Aspect Oriented Programming) to insert specialized timestamping and logging code into generic n-tier benchmarks such as RUBBoS. As application request and response messages pass from one server to another, their IDs are timestamped and logged as events for posterior analysis. Milliscope uses specialization to piggyback the timestamp and event data on each servers own logging facilities, incurring minimal logging overhead (a few percent of CPU, memory, and disk I/O for all servers in RUBBoS while recording each

request/response at every server).

- A semi-automated analytic tool, still being developed, will help its users analyze the big data to be generated by the first two components. In the past, the process of discovering millibottlenecks heavily relies on the expertise of our team in finding relevant events that coincide with the millibottlenecks. The third component will facilitate research on millibottlenecks for our team and the distributed systems research community.

CHAPTER III

AN APPROACH TO DETECT SYSTEM LEVEL MILLIBOTTLENECKS

In the chapter, we first explain why monitoring locks can help diagnosing system level millibottlenecks. we then introduce the implementation of the fine-grained lock monitoring tool based on lockstat and our three-stage approach to detect system level millibottlenecks by using this tool. Lastly, we conduct a measure of the overhead of our fine-grained lock monitoring tool and propose several solutions to reduce the overhead.

3.1 Monitor the Locks

Detecting these millibottlenecks are very challenging for three reasons: First, the kernel is a complicated system which consists of more than 10 millions of lines of code. Let's use I/O stack as an example, one file system operation has to go through various layers such as virtual file system (VFS), Ext4 file system, page cache, JBD2 journaling layer, I/O scheduler, and block layer. Second, millibottlenecks are very short, usually take tens to hundreds of milliseconds. From the sampling theorem, millibottlenecks may not be detected by monitoring tools with sampling periods of multiple seconds or minutes, which is the typical setting of many past and current performance monitoring tools. Third, revealing the root cause of millibottlenecks requires correlating to source code to the observed symptoms(VLRT requests, queue length spikes and resource utilization saturation). Understanding basic principles of systems design may not help explain why the millibottlenecks happen, but source code can provide insightful information to diagnose them. The second study case is a good example 5, the policy doesn't reveal the root cause of load balancer instability issue, but the source code does.

Numerous approaches have been proposed to understand and predict the performance of Linux kernel. There approaches designed for debugging and profiling, one example of

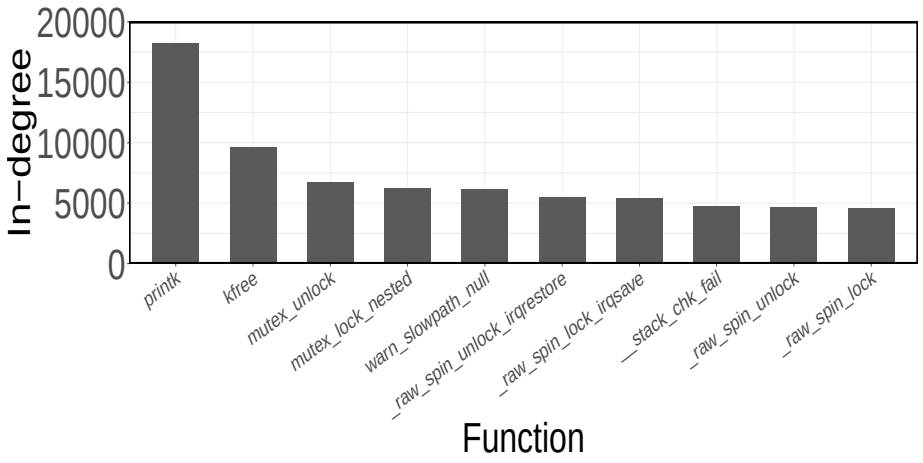


Figure 2: Top 10 in-degree functions, six of them are related to locks in the kernel.

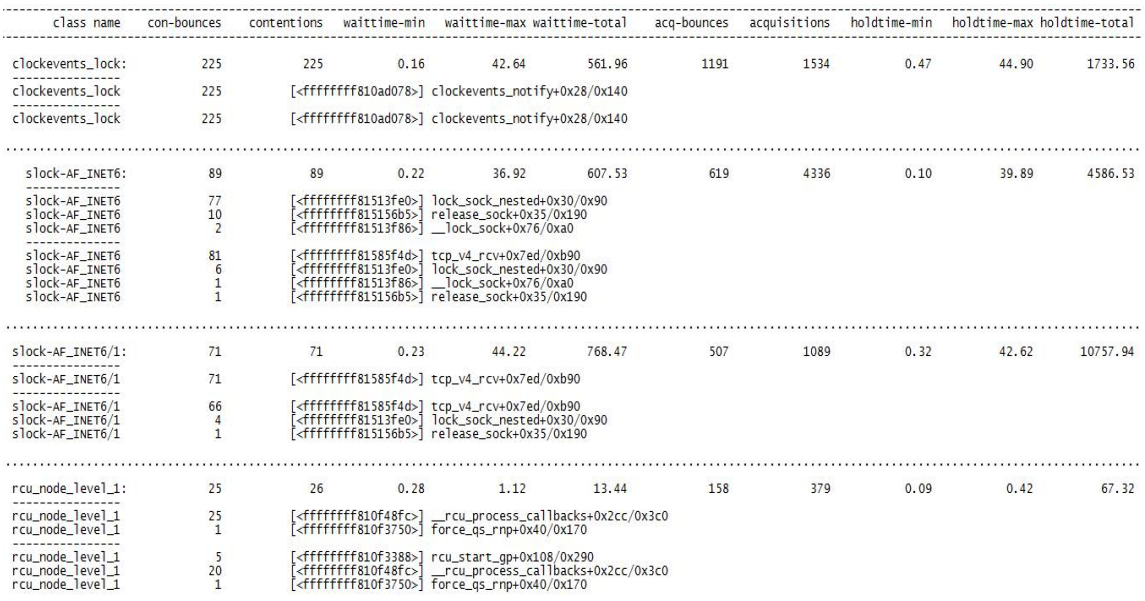
these respective approaches is DTrace [24], a facility for dynamic instrumentation of production systems. DTrace allows for many tens of thousands of instrumentation points, with even the smallest of systems offering on the order of 30,000 such points in the kernel alone. Approaches like Dtrace work by patching entry points of functions with trampolines to instrumentation routines. However, the overhead of trampolines and context switching to instrumentation routines is prohibitive for fine-grained instrumentation as simple as instruction counting. Dynamic binary translation (DBT) is a powerful technique that enables fine-grained monitoring and manipulation of an existing program binary. Feiner et al. [33] developed a DBT framework that runs as a Linux kernel module, based on the user-level DynamoRIO framework. Different from approaches like DTrace, it controls all kernel execution, including interrupt and exception handlers and device drivers, enabling comprehensive instrumentation of the OS without imposing any overhead on user-level code.

Diagnosing millibottlenecks requires identification of which resource and saturated for how long. The resource is not only limited to hardware resources such as CPU, memory, network and disk, but can also be software resource. In Linux kernel, locks are used to synchronize access to a shared resource. Compared to other software resource, locks are more fine-grained. For example, let's look at the source code of inode, which is a data structure describes a filesystem object such as a file or a directory. Multiple locks are used to protect different parts in inode [14]: inode $\rightarrow$ i_lock protects inode $\rightarrow$ i_state and inode

→ i_hash; ionode LRU list locks protect inode → i_sb → s_inode_lru and inode → i_lru; inode → i_sb → s_inode_list_lock protects inode → i_sb → s_inodes and inode → i_sb_list; bdi → wb.list_lock protects: bdi → wb.b_dirty,io,more_io,dirty_time and inode → i_io_list; inode_hash_lock protects: inode_hashtable and inode → i_hash.

We want to demonstrate monitoring locks can achieve high coverage and minimize the number of instrumenting points compared to other instrumentation based performance diagnose tools. To find the optimal instrumenting points, we generate a call graph for Linux kernel by using GCC and egypt [13]. The egypt tool takes the intermediate code generated by GCC as input and outputs the corresponding Graphviz files, which can be used to graphically represent the call graph. The call graph is a directed graph that represents the calling relationship between functions in Linux. In a call graph, there are two types of node degrees: out-degree and in-degree. Out-degree and in-degree represent the number of edges that start from the node and end at the node respectively. We focus on in-degree of a node, which indicates the number of functions that call the function corresponding to the node [50]. In the top 10 in-degree functions, six of them are related to locks in the kernel. Although there are several different types of kernel locks(such as spinlock, mutex etc), they share the same underlining implementations(lock_acquire, lock_release). These are where lockstat [16] instruments code to gather lock information. In the following slides, I also analyzed the reachability to lock_acquire. Reachability refers to the ability to get from one vertex to another within a graph. There are 94715 nodes can reach to lock_acquire node, that is 85% of total nodes can reach to lock_acquire node. Intuitively, instrumenting around the locks is reasonable because locks are used to protect a critical section of code which accesses a resource such as a data structure or device, serializing access to the resource.

lockstat provides the following statistics: number of lock contention that involved x-cpu data, number of lock acquisitions that had to wait, shortest (non-0) time we ever had to wait for a lock, longest time we ever had to wait for a lock, total time we spend waiting on this lock, average time spent waiting on this lock, number of lock acquisitions that involved x-cpu data, number of times we took the lock, shortest (non-0) time we ever held the lock, longest time we ever held the lock, total time this lock was held, average time this lock was



```
------------------------------------------------------------------------------------------------------------------------------------------------------------
       class name    con-bounces    contentions   waittime-min   waittime-max waittime-total    acq-bounces   acquisitions   holdtime-min   holdtime-max holdtime-total
------------------------------------------------------------------------------------------------------------------------------------------------------------

clockevents_lock:           225            225           0.16          42.64         561.96           1191           1534           0.47          44.90         1733.56
---------------
clockevents_lock            225          [<ffffffff810ad078>] clockevents_notify+0x28/0x140
---------------
clockevents_lock            225          [<ffffffff810ad078>] clockevents_notify+0x28/0x140

.............................................................................................................................................................

  slock-AF_INET6:           89             89           0.22          36.92         607.53            619           4336           0.10          39.89         4586.53
  --------------
  slock-AF_INET6            77          [<ffffffff81513fe0>] lock_sock_nested+0x30/0x90
  slock-AF_INET6            10          [<ffffffff815156b5>] release_sock+0x35/0x190
  slock-AF_INET6             2          [<ffffffff81513f86>] __lock_sock+0x76/0xa0
  --------------
  slock-AF_INET6            81          [<ffffffff81585f4d>] tcp_v4_rcv+0x7ed/0xb90
  slock-AF_INET6             6          [<ffffffff81513fe0>] lock_sock_nested+0x30/0x90
  slock-AF_INET6             1          [<ffffffff81513f86>] __lock_sock+0x76/0xa0
  slock-AF_INET6             1          [<ffffffff815156b5>] release_sock+0x35/0x190

.............................................................................................................................................................

slock-AF_INET6/1:           71             71           0.23          44.22         768.47            507           1089           0.32          42.62         10757.94
---------------
slock-AF_INET6/1            71          [<ffffffff81585f4d>] tcp_v4_rcv+0x7ed/0xb90
---------------
slock-AF_INET6/1            66          [<ffffffff81585f4d>] tcp_v4_rcv+0x7ed/0xb90
slock-AF_INET6/1             4          [<ffffffff81513fe0>] lock_sock_nested+0x30/0x90
slock-AF_INET6/1             1          [<ffffffff815156b5>] release_sock+0x35/0x190

.............................................................................................................................................................

rcu_node_level_1:           25             26           0.28           1.12          13.44            158            379           0.09           0.42          67.32
---------------
rcu_node_level_1            25          [<ffffffff810f48fc>] __rcu_process_callbacks+0x2cc/0x3c0
rcu_node_level_1             1          [<ffffffff810f3750>] force_qs_rnp+0x40/0x170
---------------
rcu_node_level_1             5          [<ffffffff810f3388>] rcu_start_gp+0x108/0x290
rcu_node_level_1            20          [<ffffffff810f48fc>] __rcu_process_callbacks+0x2cc/0x3c0
rcu_node_level_1             1          [<ffffffff810f3750>] force_qs_rnp+0x40/0x170
```

Figure 3: Example of lockstat output, the lock statistics consist of two parts: the actual stats and the contention points.

held. These numbers are gathered per lock class, per read/write state (when applicable). It also tracks 4 contention points per class. A contention point is a call site that had to wait on lock acquisition. Figure 3 shows the first four lock class statistics. let's take slock-AF_INET6 as an example, we see three recorded contention points (the code which tries to get the lock): lock_sock_nested, release_sock and __lock_sock. It also illustrates the four recorded contended points (the lock holder): tcp_v4_rcv, lock_sock_nested, __lock_sock and release_sock.

Using lockstat seems a promising approach to detect Linux kernel millibottlenecks, but the challenge we face is that Lockstat is cumulative, coarse-grained monitoring, it is not sufficient to pinpoint the millibottlenecks. We extended lockstat to support fine-grained timescale granularity, such that we can link the fine-grained monitoring data from resource monitoring tools to information about requests dependencies and causality. The lockstat file is available in the /proc filesystem. Like most resource monitoring tools use the shapshot /proc/stat file in Linux to estimate the CPU utilization in each monitoring interval, we obtain fine-grained lock contention data by taking the shapshot of lockstat file periodically. Like /proc/stat, the data in lockstat file is are cumulative from the time that the machine

is booted. Unlike /proc/stat, lockstat also contains minimum and maximum values for lockstat statistics. To retain these values, we have to reset the counters in lockstat after copying them to a target buffer instead of calculating the difference between two consecutive snapshots. To precisely pinpoint the time when lock statistic is obtained, we need a fine-grained timestamping method to correlate with other fine-grained resource and event monitoring data. We start timestamping when we poll the /proc/lockstat file. Our lock monitoring tool periodically polls and reset lockstat data, the default monitoring interval is 50 ms, which is small enough to capture the kernel millibottlenecks.

We apply a few technologies to reduce the overhead incurred by our lock monitoring tool. First, to reduce the amount of small and frequent write operations, we allocate enough buffers to hold all the lockstat snapshots in memory. After the experiment, we then flush the in-memory shapshots to disk. Second, we remove unnecessary information in the snapshot to reduce the amount of buffers allocated for them. As shown in Figure 3, the original snapshot file uses space and dotted line to separate lock statistics, making it more human readable. Since we build a parser to extract the lock statistics from lockstat snapshot, we use a single delimiter such as "," to separate records in a line instead of multiple blank spaces to save space. We also remove the dotted lines when we gather the lockstat snapshot to further save space. On average, we can reduce the snapshot size by 85% by through eliminating unnecessary information.

The fine-grained lock monitoring data can be used to diagnose the millibottlenecks in Linux kernel. First, lock contention, where we observe long lock hold and wait times, can help us to pinpoint the root cause of millibottleneck. Contention on the lock can limit the performance benefits from multithreading, thus results in a performance bottleneck and underutilized resources. Second, long lock hold time but no contention can also help us to diagnose the root cause the millibottleneck. lock hold time indicates how long the resource protected by the lock is used by a single thread, which is a more fine-grained source monitoring in Linux kernel. For example, resource monitoring tools like sar or collectl don't tell us which process is using I/O resource when we observe millibottlenecks caused by I/O saturation. But lockstat can provide more detailed information: which locks are being used

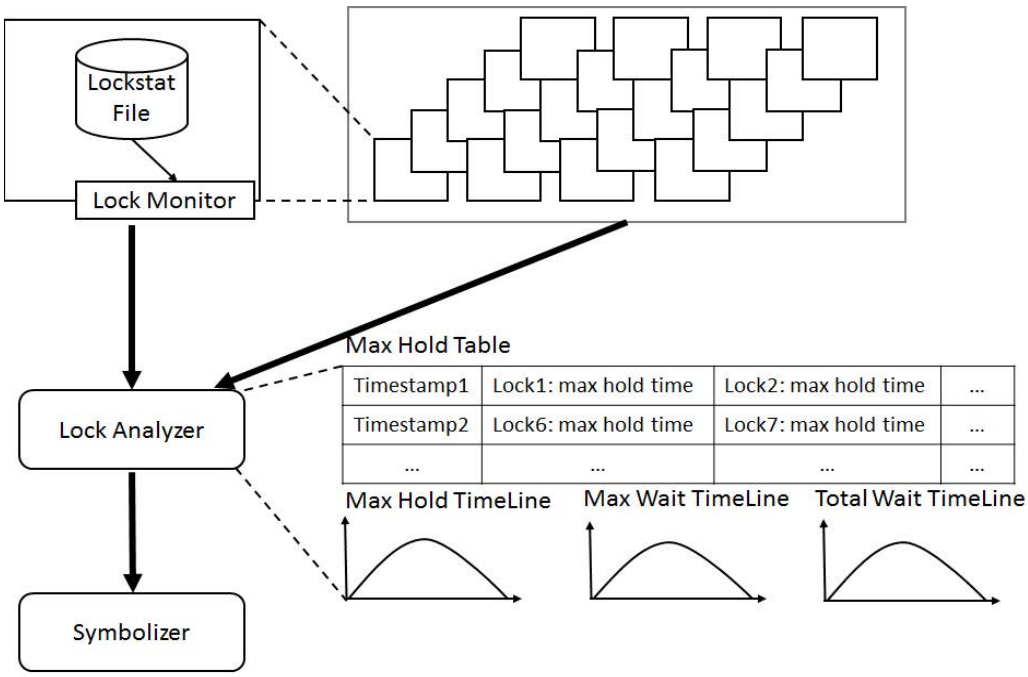


Figure 4: High-level overview of detecting kernel millibottlenecks pipeline: first, we use our lock monitoring tool to collect fine-grained lock statistics. Second, we identify the contended locks that cause millibottlenecks. Third, we reveal the root causes of the lock contentions.

to access the I/O resources and where in the kernel these locks are being held for excessive periods of time. Examining this information can help us to identify places in the kernel where the code may cause the millibottlenecks.

3.2 Detect Kernel Millibottlenecks

If we obverse one CPU core is saturated, while the remaining cores are idle, it is possible that the millibottleneck is caused by lock contention. The Linux kernel millibottleneck detection pipeline is a three-stage process (see Figure 4). In the first stage, we run RUBBoS benchmark with the MilliMonitor tools and our lock monitoring tool to find when and where the millibottlenecks happen. We analyze the point-in-time response time graph to find when the VLRT requests happen. We then use all servers' event data gather by MilliMonitor tools find the queues in each server during the millibottleneck period(s). The server with long queue indicates that the server is suffering from millibottlenecks.

In the second stage, we search the contended locks to identify the one that causes the VLRT requests. The second stage involves two steps: (1) We narrow the search space to

a set of candidates: we zoom into the millibottleneck period, sort the locks by max wait time, max hold time, total wait time, total hold time, and then pick the top 3 candidates. (2) We plot the candidates' max hold time, max wait time and total wait time timeline graph, and then correlate them with VLRT and queue length graph. We start with top 1 contended lock, If they do not bear any relationships, then the current lock is discarded and this step is repeated. Note that the lock hold/wait time may be greater than the monitoring interval(50ms) and lockstat assigns the window number based on release timestamp, which is not aligned with the accounting approach used in our study of millibottlenecks. We can infer lock acquire time window based on max hold time and lock release time windows: $Acquire\quad time\quad window = release\quad time\quad window - max\quad hold\quad time$, such that we can adjust to release-timestamp-based accounting approach.

In the last stage, to reveal the root causes of millibottlenecks, the VLRT requests must be correlated with the source code. The contended lock and the contention point can bridge the gap between the source code and the VLRT requests. Recall we have built the execution graph of Linux kernel in section 3.1, we start from the contented point(the lock holder), forward slice to find all functions that causally and transitively depend on the contended point. The symbolization process is very complicated, it involves an iterative process of correlating with fine-grained resource and event monitoring data.

3.3 Measurement Overhead

In this section, we present the overhead of lock monitoring tool. Lockstat overhead is the most critical segment of lock monitoring performance footprint. Although lockstat offers a kernel lock monitoring capability that is very powerful and easy to use, it itself perturbs the system performance by adding measurable overheads. Our experiments use 1 Apache server, 1 Tomcat server, and 1 MySQL server. We only ran Lock Monitor on Mysql server under two different workload modes: browse-only and read/write mixes. The Mysql server is running on Linux 2.6.43 on an 8-core Intel Xeon machine. Figure 5 show the CPU utilization of Mysql server or workloads from 10000 to 30000 concurrent clients, the average CPU utilization of MySQL rise gradually, as expected. We also observe Lock Monitor

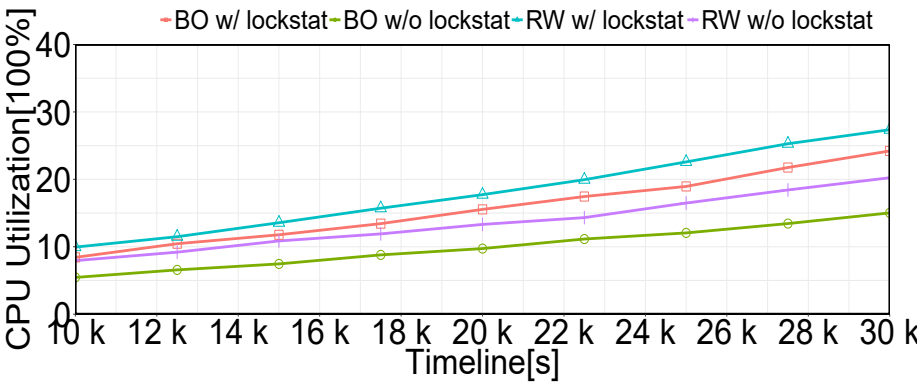


Figure 5: Lock monitoring overhead under various workloads. Lockstat incurs non-negligible overhead and the overhead increases as workload increases.

overhead increases as workload increases: for the browse only workload, the Lock Monitor uses CPU resource ranging from 3.0% to 9.2%; for the read write mix workload, the Lock Monitor uses CPU resource ranging from 1.0% to 6.0%. As expected, more lock events happen when the workload increases.

In order to avoid noticeable latency degradation, we will adopt sampling mechanism in the further work. Sampling can reduce the overhead and maintain the effectiveness since the vast majority of interest were still very likely to appear often enough to be captured. For instance, a sample of just one out of hundreds of lock events may provide sufficient information for many common uses of the tracing data.

3.4 Conclusion

In the section, we propose a novel approach to detect system level millibottlenecks by fine-grained monitoring of locks. Firstly, we demonstrate monitoring locks can achieve high coverage of source code and fine-grained resource monitoring because locks are used to protect a critical section of code which accesses a resource such as a data structure or device, serializing access to the resource. Secondly, we implement a fine-grained lock monitoring tool based on lockstat. Lastly, we evaluate the overhead of our lock monitoring tool.

CHAPTER IV

STUDY CASE 1: LINUX KERNEL MILLIBOTTLENECKS

In this section, we present a case study of an especially complex system level millibottleneck caused by conservative stable page. To better understand the issue, we first provide background on CPU and I/O schedulers in Linux. We then show concrete experimental evidence of the latency long-tail problem using a standard n-tier web application benchmark when Linux is flushing dirty pages. Lastly, we found the root cause of the millibottlenecks by using our fine-grained lock monitoring tool and provide remedies to eliminate the millibottleneck.

4.1 Background: CFS & CFQ

Completely Fair Scheduler(CFS) was introduced to Linux in kernel 2.6.23 [3] as the default CPU scheduler. CFS approximates an "ideal, precise multi-tasking CPU" on real hardware, it tries to mimic perfectly fair scheduling. CFS maintains the amount of time provided to a given task in what's called the virtual runtime, which is actual runtime normalized to the total number of running tasks. The smaller a task's virtual runtime means the higher its need for the processor. CFS uses red-black tree to maintain runnable processes: all nodes with virtual runtime are stored on the left side of the root, while those who have got larger time on processor are stored on the right side. Whenever kernel wants to select next process for scheduling, it takes the left most node of red-black tree and grants processor to it. Red-black tree has two advantages over other data structure: first, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other. Second, operations on the tree are very efficient, the time complexities of insertion and deletion are O(log n) time (where n is the number of nodes in the tree).

Completely Fair Queuing(CFQ) is the default I/O scheduler in Linux [2]. The goal of CFQ is to equally distribute I/O bandwidth equally among processes performing I/O operations. CFQ has separate queues for synchronous I/Os and asynchronous I/Os. it maintains the per process queue for the processes which request I/O operation(synchronous requests)

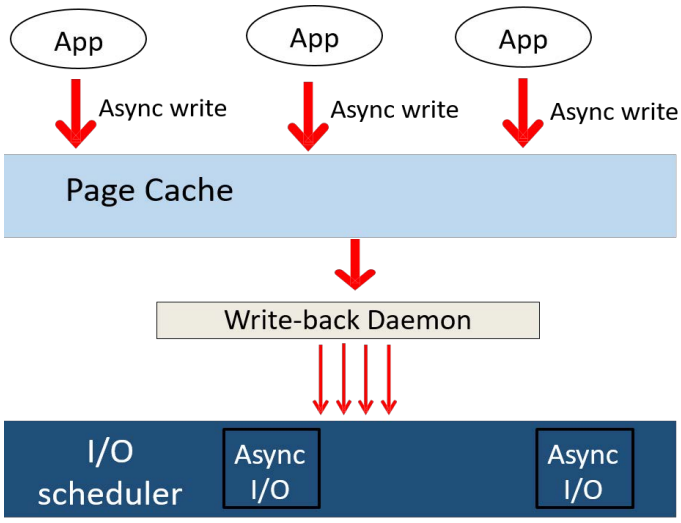


Figure 6: Write delegation: Linux performs write operations directly into the page cache instead of immediately to the disk, then these write requests are forwarded to the I/O scheduler by writeback daemon processes

and services these queues in round robin fashion and selects a few requests (default 4) from each queue. In the case of asynchronous requests, all the requests from all the processes are batched together based on their process's I/O priority. Since most asynchronous I/Os are submitted by the write daemons process(I/O priority 4), they are put into the same asynchronous CFQ queue. All the synchronous queues have higher priority than asynchronous queues in CFQ [35].

To minimize disk I/O, Linux performs write operations directly into the page cache instead of immediately to the disk, this strategy is called writeback. The synchronous write is shown in Figure 6. The written-to pages in the page cache are marked as dirty. Periodically, the dirty pages are written back to disk. Linux uses daemon process called pdflush to writes data out of the page cache. The parameters listed below affect the frequency that dirty pages written back to disk.

- dirty_background_ratio: a percentage of total available memory that contains free pages and reclaimable pages, the number of pages at which the pdflush threads will start writing out dirty data.

- dirty_ratio: a percentage of total available memory that contains free pages and reclaimable pages, the number of pages at which a process which is generating disk writes will itself start writing out dirty data

- dirty_writeback_centisecs: How often the pdflush threads wake up and write dirty pages out to disk.

- dirty_expire_centisecs: it defines when dirty data is old enough to be eligible for write back to the disk.

The limitation of CFQ scheduler is it ignores priorities of synchronous writes. First, CFQ scheduler doesn't allocate memory bandwidth fairly between processes based on their priorities when they write to the page cache. Second, the writeback daemon process doesn't take priorities into consideration, it searches for dirty pages in the radixtree and flushes them to disk when free memory drops those thresholds. Our observation is reinforced by the study of [55]. They found the writeback task sometimes appears responsible for all write traffic [55] and the write task ignores the priorities of the asynchronous queues. Yang et al. evaluated CFQ using an asynchronous write workload, they evaluated write performance of eight threads with different priorities. They found CFQ ignores priorities, treating all threads equally. The reason is that all the requests appear to have a priority of 4 to CFQ, because the writeback thread (a priority-4 process) submits all the writes on behalf of the eight benchmark threads.

4.2 VLRT Requests Appear When Linux is Flushing Dirty Pages

We follow the 5-step approach described in section [45] to study the millibottleneck. In the first step, we run the RUBBoS benchmark with the MilliMonitor tools (resource and event monitors) activated to collect fine-grain resource and event data. we used the simplest configuration (1 Apache server, 1 Tomcat server and 1 MySQL server).

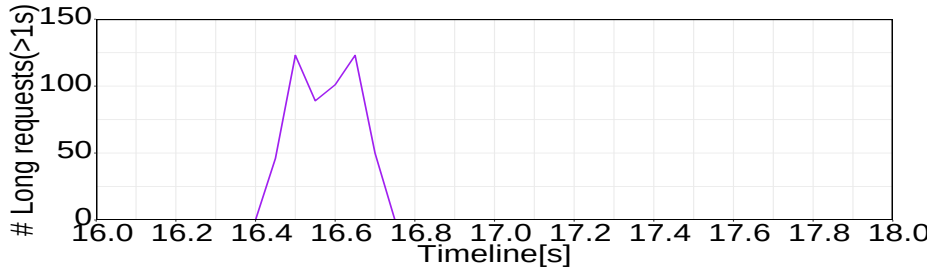
In the second step, we analyze the point-in-time response time graph to find the long requests. Figure 7(a) shows a 2-second time interval in which one cluster of VLRT requests can be seen (at approximately 16.4 seconds from the beginning of the experiment). Once the

long request peaks are detected, we use the MilliMonitor resource monitors (first component) to search for millibottlenecks in resources being monitored.

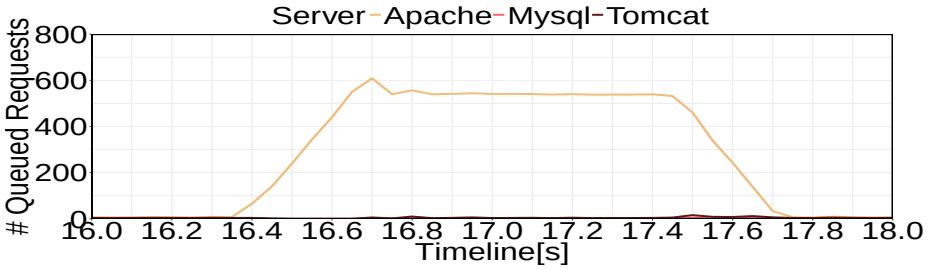
In the third step, we analyze the event data from all servers to find the queues in each server during the millibottleneck period(s). Since the long requests are due to performance bugs, one or more servers must have queues. Figure 12(a) shows queued requests in Apache, Tomcat and MySQL during this same time interval. We use queue length graph to determine if there are millibottlenecks: large spikes in the graph represent an abnormally large number of queued requests, which from our experience are usually indicative of bottlenecks. On the other hand, the steady short queue length of a tier (or server) might indicate the tier (or server) is performing well. We see a strong correlation between peaks of Apache's queue length in Figure 12(a) and peaks in VLRT requests in Figure 7(a). A detailed analysis of the Apache queues (omitted here) [53] shows that packets are being dropped by Apache due to queue overflow. VLRT requests materialize due to dropped packets being retransmitted. The Apache queue peak indicates the existence of the millibottleneck in Apache.

In the fourth step, we turn resource monitoring on new target resources, either based on expert experience as we have done so far, or using the suggested targets from the analysis performed by MilliMonitor. Further analysis of the queues and resource utilization of Apache, Tomcat, and MySQL shows that the Apache CPU saturation causes the millibottlenecks that lead to the observed elongated queues, as shown in Figure 7(c).

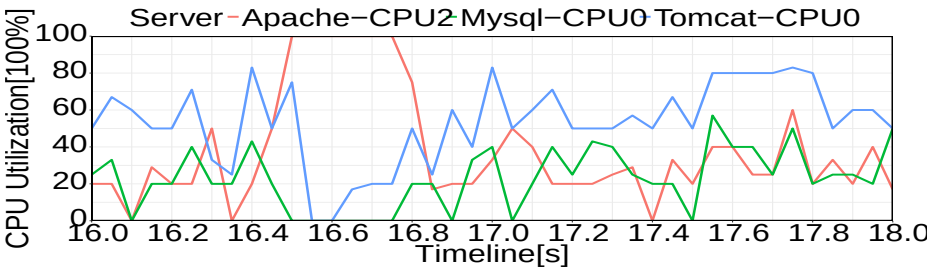
In the final step, we identify and study the software and hardware components involved in the formation of millibottlenecks. We found pdflush contributed 100% iowait by using Iotop [15]. Pdflush is a process who is responsible for writing back dirty pages in page cache to disk. These dirty pages mainly are Tomcat logs, which include access, servelet and localhost logs. We show the sum of dirty pages changes in Figure 7(f), We see the abrupt drops correlate strongly with iowait saturations in Figure 7(d), which confirms flushing dirty pages lead to transient CPU saturations in Figure 7(c). This is an unexpected result: flushing dirty pages should have minimal impact on foreground tasks since it is supposed to be asynchronous.



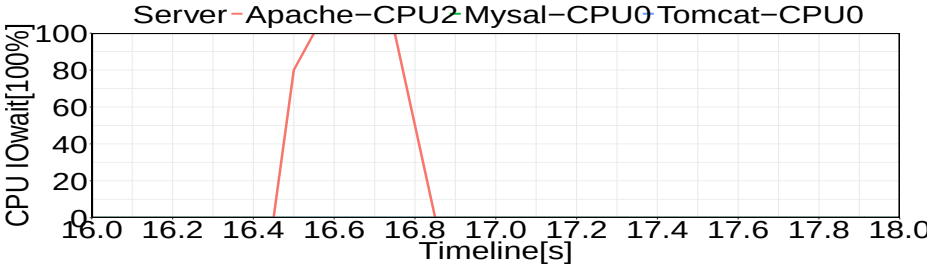
(a) Number of VLRT (>1000ms) requests counted at every 50ms time window.



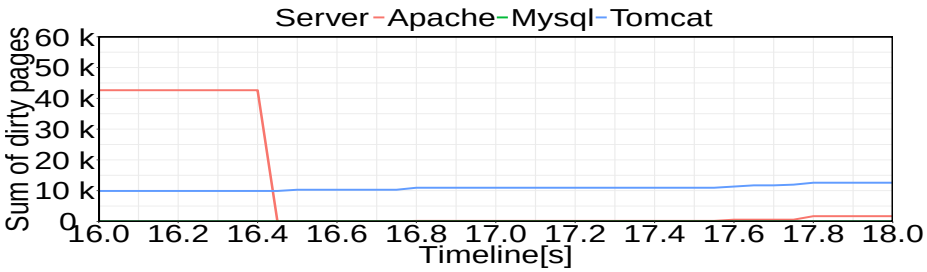
(b) Request queue for Apache, Tomcat and MySQL tier, the queue peaks match well with the occurrence of the VLRT requests.



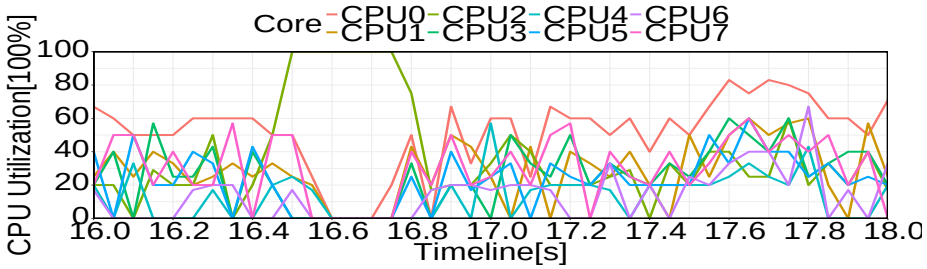
(c) Transient CPU saturations of Apache servers correlates with the queue spikes in the corresponding Apache server.



(d) Iowait saturations correlates with Transient CPU saturations, indicating I/O activities cause CPU saturations.



(e) Abrupt dirty page cache size drops correlate with iowait saturations, suggesting flushing dirty pages cause iowait saturations.



(f) One core is saturated while the remaining cores are idle.

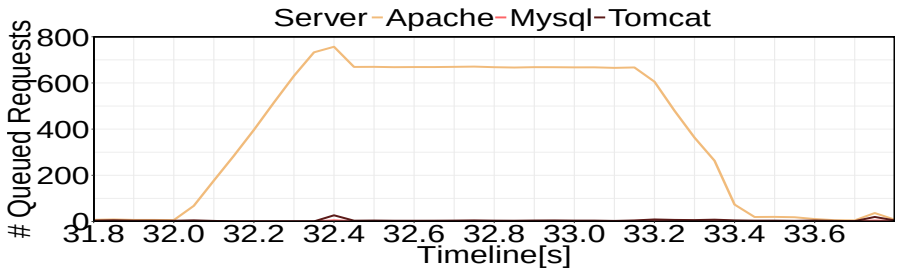
Figure 7: VLRT requests appear when Linux is flushing dirty pages.

4.3 Priority In Linux: From CPU to IO

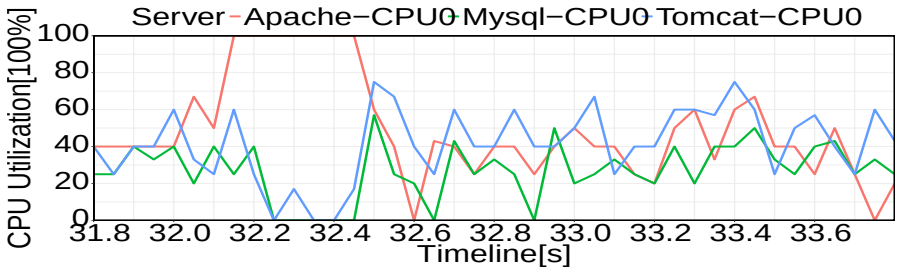
4.3.1 Boost CPU priority

Writeback daemon processes are supposed to have minimal impact on foreground process. We first suspect the millibottlenecks are caused by priority inversion [7] in CPU scheduling, where a high priority task cannot make any progress as a low priority task has a resource it requires. The kernel uses a scale from 0 to 139 to represent priorities internally. Priorities 0-99 for real-time processes. Priorities 100-139 for normal processes, users can change normal process's priority value via nice() system call. By default, a process's priority is 120, by setting nice value to -20 +19, the priority value of a normal process is mapped into 100 - 139.

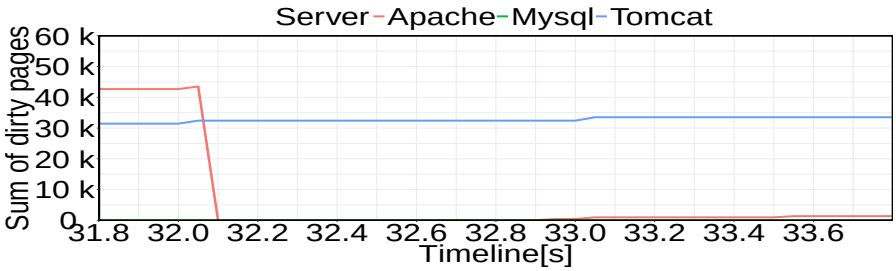
By default, HTTPD, Tomcat and Pdflush processes have the identical priority. We boosted HTTPDs and Tomcats priority by using nice command, namely, the processes issuing asynchronous writes have higher priority than the writeback daemon process. We see boosting priority didn't improve the average response time. We also examined the queue length and CPU utilization graph when dirty pages were being flushed out in 8. We observe huge and sharp spike in 8(a), suggesting the root cause of the millibottlenecks is



(a) Request queue for Apache, Tomcat and MySQL tier, the queue peaks match well with the occurrence of the VLRT requests.



(b) Transient CPU saturations of Apache servers correlates with the queue spikes in the corresponding Apache server.



(c) Abrupt dirty page cache size drops suggest Linux is flushing dirty pages.

Figure 8: Boosting CPU priorities fails to lower the peaks in queue length graph, suggesting the millibottlenecks are not caused by CPU scheduler priority inversion.

not priority inversion in CPU.

4.3.2 Boost IO priority

Our second hypothesis is millibottlenecks are caused by priority inversion in IO scheduling. To valid or invalid our hypothesis, we need to understand how priority is calculated. Firstly, the I/O priority of a process can be assigned by CFQ scheduler. CFQ scheduler implements three generic scheduling classes that determine how I/O is served for a process [1]:

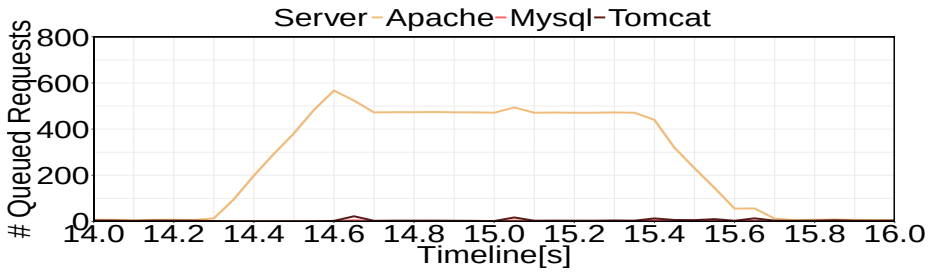
- Idle: A program with idle I/O priority will only get I/O resource when no other program has asked for I/O resource for a defined grace period. The impact of an idle

I/O process on normal system activity should be zero.

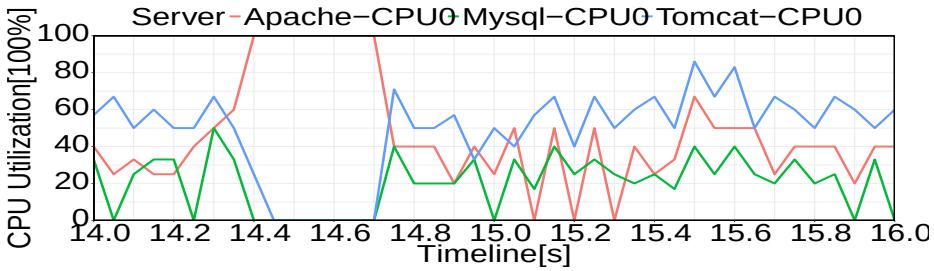
- Best-effort: This is the effective scheduling class for any process that has not asked for a specific I/O priority. This class takes a priority argument from0-7, with a lower number being a higher priority. Programs running at the same best-effort priority are served in a round-robin fashion.
- Realtime: The RT scheduling class is given first access to the disk, regardless of what else is going on in the system. Thus the RT class needs to be used with some care, as it can starve other processes. As with the best-effort class, 8 priority levels are defined denoting how big a time slice a given process will receive on each scheduling window.

Secondly, the I/O priority can also be derived from the CPU nice level of the process. For kernel version after 2.6.26, a process that has not asked for an I/O priority inherits its CPU scheduling class:$io\_priority = (cpu\_nice + 20)/5$. IO priority ranges from 0 to 7 with 0 being the highest priority, while CPU niceness ranges from -20 to 19. Different CPU priorities could be mapped to the same IO priority, which means the I/O priority precision is lost when derived from the CPU priority. But our experimental results show that I/O priority is not derived from the CPU priority. We change a processs CPU nice level from -20 to 19, but the processs I/O priority remains 4.

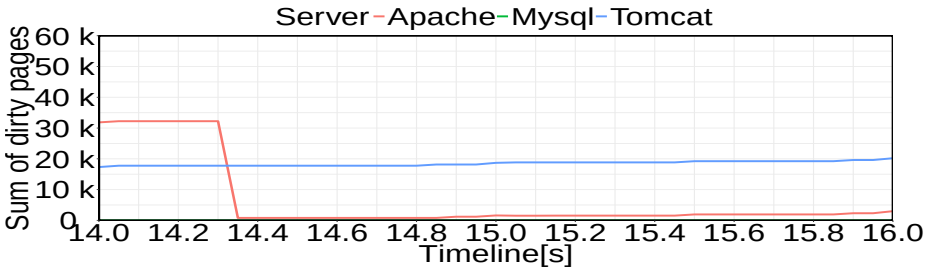
We boosted HTTPDs and Tomcats I/O priority by using ionice command, but didn't see an improvement in average response time. As shown in Figure 9, we also examined the queue length and CPU utilization graph when dirty pages were being flushed out in 8. We observe huge and sharp spike in 8(a), suggesting boosting the I/O priority of the process issuing write request can't eliminate the millibottlenecks. This can be explained by CFQ scheduler ignores priorities of asynchronous writes, the write-back daemon process does the I/O work on behalf of other processes. Our observations are reinforced by a recent study in [55]. The limitations of CFQ scheduler alone doesn't create the millibottlenecks, there must exist a complicated dependency between the process issuing asynchronous writes and the writeback daemon process.



(a) Request queue for Apache, Tomcat and MySQL tier, the queue peaks match well with the occurrence of the VLRT requests.



(b) Transient CPU saturations of Apache servers correlates with the queue spikes in the corresponding Apache server.

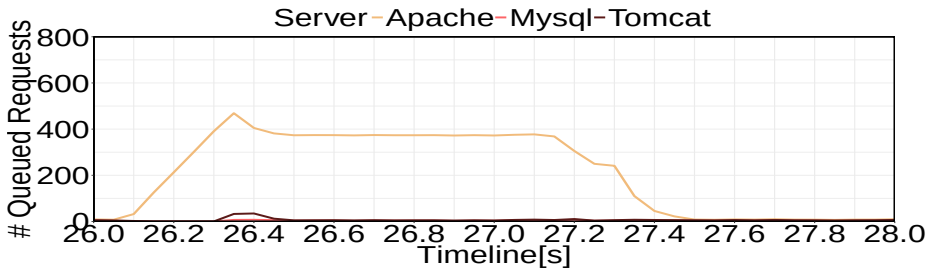


(c) Abrupt dirty page cache size drops suggest Linux is flushing dirty pages.

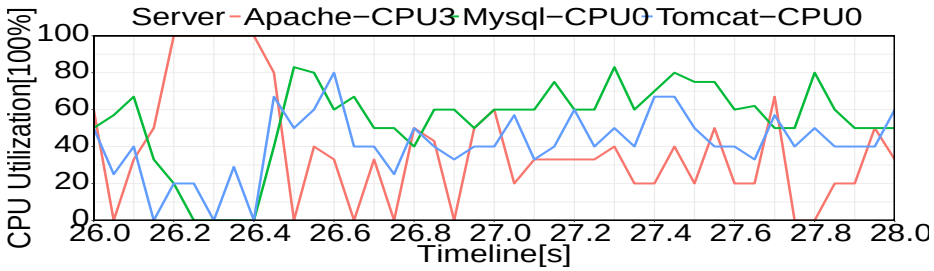
Figure 9: Boosting IO priority fails to eliminate the millibottlenecks, suggesting CFQ scheduler ignoring IO priorities alone doesn't create the millibottlenecks.

4.4 Comparison with Perf

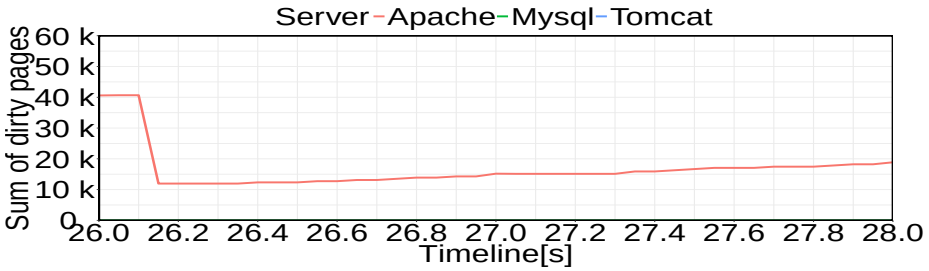
To have a better understanding the benefits of our approach of diagnosing system level millibottlenecks, we compare it against perf, a profiler tool for Linux that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface [5]. Perf tool offers a rich set of commands to collect and analyze performance and trace data, in this thesis, we focus on two of the most commonly used ones. Since the performance counters in perf are cumulative, to support fine-grained monitoring, we used the same approached as we used for lockstat, namely, we periodically execute perf command with fine-grained monitoring interval(default value is 50ms). To correlate with other



(a) Request queue for Apache, Tomcat and MySQL tier, the queue peaks match well with the occurrence of the VLRT requests.



(b) Transient CPU saturations of Apache servers correlates with the queue spikes in the corresponding Apache server.



(c) Abrupt dirty page cache size drops suggest Linux is flushing dirty pages.

Figure 10: Zoom into a small period when millibottlenecks happen and use perf record command to diagnose the root cause of the millibottlenecks

fine-grained resource and event monitoring date, We record timestamp when we run perf command. We first start with perf record command, which gathers a performance counter profile and saves into a binary file called, by default, perf.data.

By default, perf record uses the cycles event as the sampling event. We run RUBBoS benchmark with the MilliMonitor tools and perf record command at its default rate(1000Hz, or 1000 samples/sec). We use 1 Apache server, 1 Tomcat server, and 1 MySQL server at workload 15000 clients. After collecting all the fine-grained resource and event data, we then use perf report command to read files generated by perf record command and output concise execution profiles. By default, samples are sorted by functions with the most samples first.

We provide top 5 functions with the most samples in each 50ms time window during the millibottleneck period in table 3. Showing CPU cycles is not adequate to explain why HTTPD processes are stuck when Linux was flushing dirty pages.

Perf record command doesn't yield a diagnosis, we attempt to use perf lock command to investigate the issue. To enable perf lock command, we need to recompile the kernel with the same configuration as enabling lockstat. We think perf lock is a consumer that uses the lockstat provider to gather its raw data. We first periodically collect perf with time interval 50ms. The servers are frozen due to running out of memory, such that we change the monitoring interval to 1 second. This demonstrates that shortcoming of perf lock, it incurs more memory overhead than our tools.

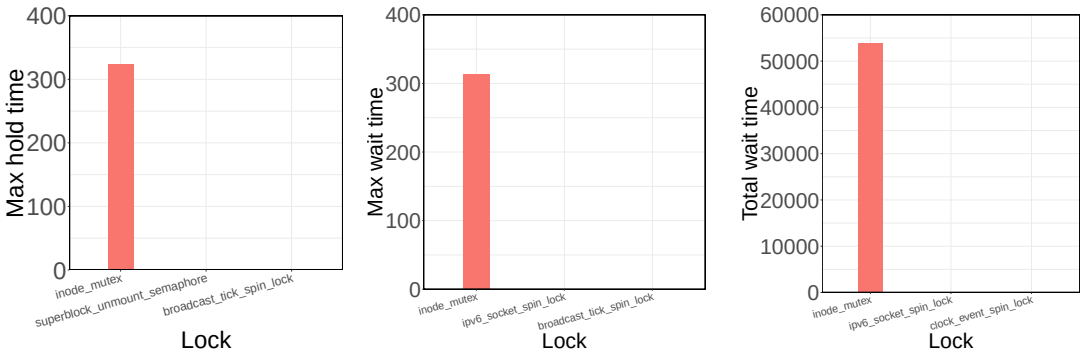
4.5 Root Cause Analysis

In this section, we use MilliMonitor and our fine-grained lock monitoring tools to reveal the complicated dependency. Recall that in the second stage of diagnosing system level millibottleneck, we need to correlate the contended with the VLRT requests. First, we zoom into the critical interval during which the Apache queue length has a huge spike, and then sort the locks by max wait time, max hold time, total wait time, total hold time. Figure 11 shows the top 3 contended candidates, the most contended lock is inode mutex, which allows a single thread at a time to access the inode. Compared to inode mutex, the max hold time, max wait time and total wait time of the second and third most contended lock are negligible. Second, we plot the candidates' max hold time, max wait time and total wait time timeline graph in figure 12, we found contention on inode mutex is strongly correlated with the huge spike in Apache queue length, confirming that the lock contention on inode mutex causes VLRT requests.

To reveal the root cause of the lock contention, we must correlate it to the source code. We build the dependency graph, figure 13 shows the critical parts of it. We forward slice from the contention point(generic_file_aio_write), inspect every function that causally and transitively depends on the contended point. Combined with the contextual information, Linux was flushing dirty pages in the critical interval, we confirmed wait_on_page_writeback

Table 3: Top 5 functions with the most samples in each 50ms time window during the millibottleneck period, but the perf result doesn't yield a diagnosis.

Period	Overhead	Command	Shared Object	Symbol
26.05-26.10s	4.60%	perf	libc-2.14.1.so	__GI_____strtoll_l_internal
	4.53%	swapper	[kernel.kallsyms]	poll_idle
	4.37%	perf	[kernel.kallsyms]	vsnprintf
	3.62%	perf	[kernel.kallsyms]	format_decode
	2.71%	perf	[kernel.kallsyms]	link_path_walk
26.10-26.15s	12.17%	swapper	[kernel.kallsyms]	intel_idle
	4.73%	perf	[kernel.kallsyms]	format_decode
	4.25%	perf	[kernel.kallsyms]	__alloc_pages_nodemask
	3.63%	perf	[kernel.kallsyms]	link_path_walk
	3.63%	perf	[kernel.kallsyms]	vsnprintf
26.15-26.20s	21.32%	swapper	[kernel.kallsyms]	poll_idle
	5.50%	swapper	[kernel.kallsyms]	intel_idle
	4.72%	perf	[kernel.kallsyms]	vsnprintf
	4.67%	perf	[kernel.kallsyms]	format_decode
	3.63%	perf	[kernel.kallsyms]	number
26.20-26.25s	18.23%	swapper	[kernel.kallsyms]	poll_idle
	6.01%	perf	libc-2.14.1.so	__memcpy_sse2
	5.65%	perf	[kernel.kallsyms]	number
	5.57%	swapper	[kernel.kallsyms]	_raw_spin_lock_irqsave
	3.50%	perf	[kernel.kallsyms]	_raw_spin_lock
26.25-26.30s	6.54%	perf	[kernel.kallsyms]	vsnprintf
	4.36%	swapper	[kernel.kallsyms]	default_send_IPI_mask_sequence_phys
	4.24%	swapper	[kernel.kallsyms]	intel_idle
	4.24%	swapper	[kernel.kallsyms]	format_decode
	3.74%	perf	[kernel.kallsyms]	system_call
26.30-26.35s	5.83%	swapper	[kernel.kallsyms]	intel_idle
	4.33%	perf	[kernel.kallsyms]	memcpy
	3.68%	httpd	httpd	0x271a8
	2.58%	httpd	libapr-1.so.0.4.5	0x1a000
	2.55%	perf	[kernel.kallsyms]	strchr
26.35-26.40s	5.30%	httpd	httpd	0x33d1d
	4.04%	httpd	libapr-1.so.0.4.5	0x19680
	3.37%	swapper	[kernel.kallsyms]	intel_idle
	3.00%	swapper	[kernel.kallsyms]	poll_idle
	2.60%	httpd	[kernel.kallsyms]	select_task_rq_fair



(a) Top 3 max hold contending locks (b) Top 3 max wait contending locks (c) Top 3 total wait contending locks

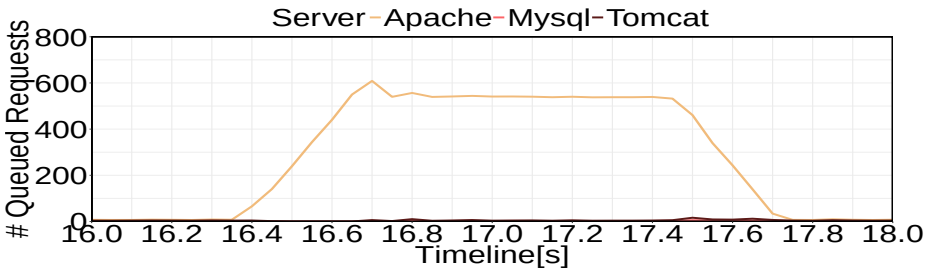
Figure 11: Identify the contending lock by comparing max hold, max wait and total wait times

is the cause of lock contention on inode mutex. wait_on_page_writeback function is used when a task wants to guarantee some data to be written back for synchronizing or truncating a file or to perform a partial write to a data page, it checks whether the previously-issued I/O has reached the storage device.

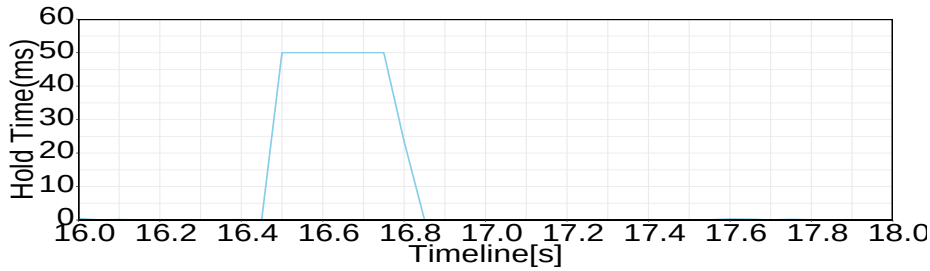
When HTTPD threads append data partially within a data page, they can be blocked since the target data page may be already flushed out asynchronously by the writeback daemon process, as shown in figure 14. The task cannot proceed its execution until the data page hits the storage. HTTPD Process1 is the holding the mutex of the inode and waiting for writeback completion. HTTPD process2 is trying to acquire the mutex, so we observe lock contention on inode mutex.

The root cause is stable page [9]: pages under writeback are marked as not being writable; any process attempting to write to such a page will block until the writeback completes. The reason that conservative stable page was invented is that modification of in-flight pages can create trouble; examples include hardware where data integrity features are in use, higher-level RAID implementations, or filesystem-implemented compression schemes. In those cases, unexpected data modification can cause checksum failures or, possibly, data corruption.

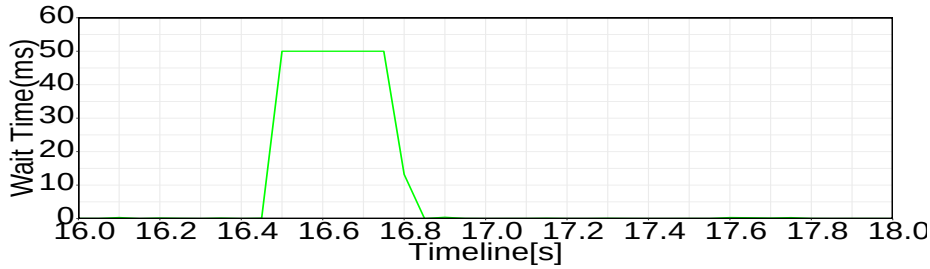
Blktrace [23] provides detailed block layer information concerning individual I/Os. it traces detailed information about request queue operations from user space to block layer



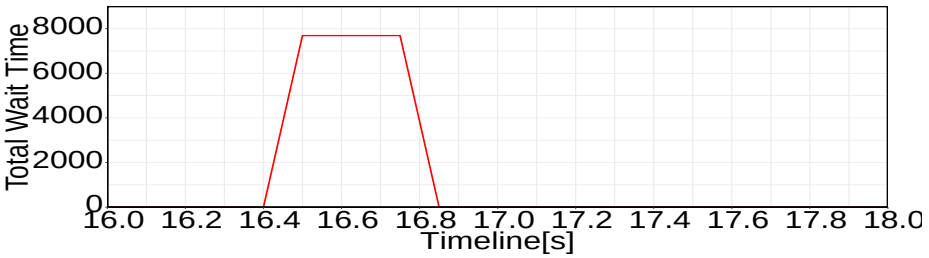
(a) Request queue for Apache, Tomcat and MySQL tier, the queue peaks match well with the occurrence of the VLRT requests. The queue peak in Apache is caused by a millibottleneck on Apache.



(b) Inode mutex max hold.



(c) Inode mutex max wait.



(d) Inode mutex total wait.

Figure 12: Inode mutex max hold, max wait and total wait time are strongly correlated with the peak in queue length graph, suggesting lock contention on inode mutex causes the VLRT requests

with timing information. Blktrace can capture events like:

- Q Request queue entry allocated.

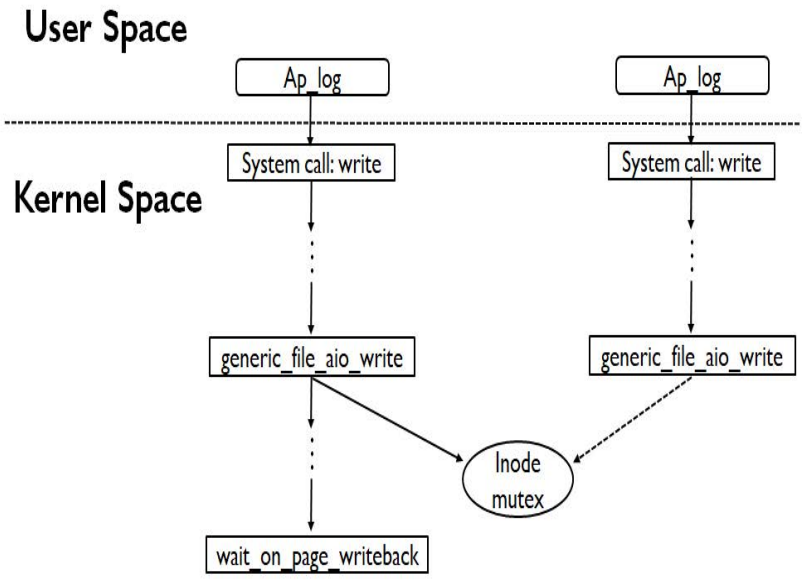


Figure 13: Function Dependency graph in Linux kernel: wait_on_page_writeback is the cause of lock contention on inode mutex

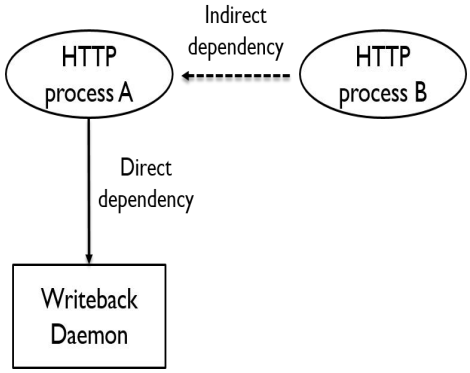


Figure 14: Direct and indirect dependency on writeback daemon process: HTTPD Process1 is the holding the mutex of the inode and waiting for writeback completion and HTTPD process2 is waiting for HTTPD Process1 releasing the mutex lock.

- G Get request.
- I Request queue insertion.
- D Request issued to the underlying block device.
- C Request completed.

In our experiments, we focus on time from a request is inserted into the device's queue to the time it is actually issued to the device($I \rightarrow D$) and service time of the request by the

device($D \rightarrow C$). We first look at the blktrace of normal flushing dirty page in table 4. We see both ($I \rightarrow D$) and ($D \rightarrow C$) are pretty short, ($I \rightarrow D$) is 21 µs and ($D \rightarrow C$) is 789 µs. In table 5, we then show the blktrace of stable write,, which can be identified by looking at the process field. We see the request was issued by write-back daemon process, then the in-flight pages were modified by HTTPD process, thus the ownership of the request was changed to HTTPD. Such trace was captured every 30 seconds, confirming that HTTPD process was modified the dirty pages. We see ($I \rightarrow D$) and ($D \rightarrow C$) dynamically increased by 1939X and 169X respectively, explaining why stable page stalled HTTPD processes.

Table 4: Blktrace for normal dirty page flush demonstrates the time spent on queue and device service time are very short.

Dev	CPU	Sequence Number	Timestamp(s)	PID	Event	RWBS	Start block + # of blocks	Process
8,4	3	936	72.506456090	682	Q	W	896809632 + 344	[flush-8:0]
8,4	3	937	72.506460778	682	G	W	896809632 + 344	[flush-8:0]
8,4	3	987	72.506819492	682	I	W	896809632 + 344	[flush-8:0]
8,4	3	1004	72.506840575	682	D	W	896809632 + 344	[flush-8:0]
8,4	3	1169	72.507629601	682	C	W	896809632 + 344	[0]

Table 5: Blktrace further confirms the millibottlenecks are caused by conservative stable page.

Dev	CPU	Sequence Number	Timestamp(s)	PID	Event	RWBS	Start block + # of blocks	Process
8,4	3	1363	72.508659714	682	Q	W	896841816 + 360	[flush-8:0]
8,4	3	1364	72.508660173	682	G	W	896841816 + 360	[flush-8:0]
8,4	3	1395	72.508802734	682	I	W	896841816 + 360	[flush-8:0]
8,4	3	1791	72.549520817	22370	D	W	896841816 + 360	[httpd]
8,4	3	1920	72.683202511	0	C	W	896841816 + 360	[0]

4.6 Remedy

In this section, we offer a remedy to improve this situation. Since our hardware does not require data integrity, conservative stable page is not necessary and does harm the performance. We adopt a Linux path to correct the situation, optimistic stable page [6],

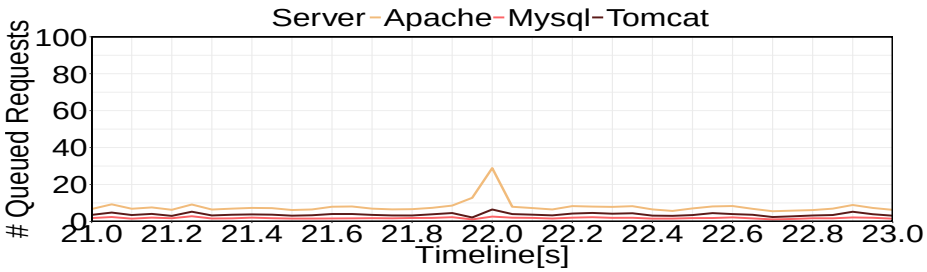
Table 6: Blktrace further confirms the effectiveness of optimistic stable page: the time spent on queue and device service time are shorted than conservative stable page.

Dev	CPU	Sequence Number	Timestamp(s)	PID	Event	RWBS	Start block + # of blocks	Process
8,4	2	1254	33.820684101	664	Q	W	212798056 + 320	[flush-8:0]
8,4	2	1255	33.820686034	664	G	W	212798056 + 320	[flush-8:0]
8,4	2	1270	33.820805252	664	I	W	212798056 + 320	[flush-8:0]
8,4	0	472	33.820909443	1731	D	W	212798056 + 320	[httpd]
8,4	0	520	33.835634986	1785	C	W	212798056 + 320	[0]

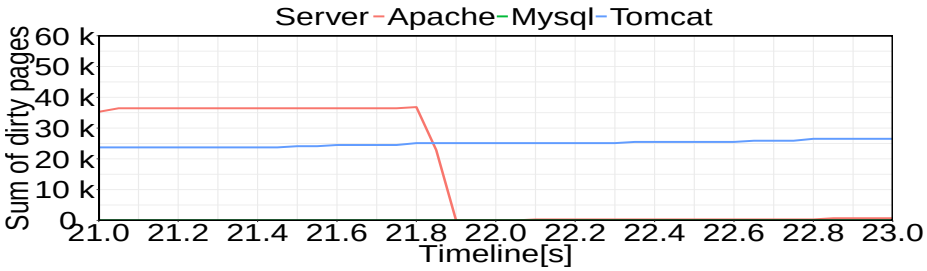
the core idea is that it checks if the given page is related to a backing device that requires page contents to be held stable during writeback. If so, then it will wait for any pending writeback to complete. we ran the same experiment on Linux 2.6.43 with optimistic stable page on the same hardware as the previous experiment. Experimental results show our remedy significantly improve average response time by 4X. To further verify the effectiveness of the optimistic stable page, we plot the queue length graph firstly in Figure 15(a). We see the queue lengths of Tomcat and Apache tier are much lower than conservative stable page. This is because optimistic stable page doesn't block Apache threads when Linux flushes dirty pages.

CPU utilization graph is a more intuitive way to verify the effectiveness of optimistic stable page. We zoom into a period during which one Apache is flushing dirty page, as shown in Figure 15(b). The CPU utilization graph is shown in Figure 15(c), we see none of the 8 cores are idle when Linux was flushing dirty pages.

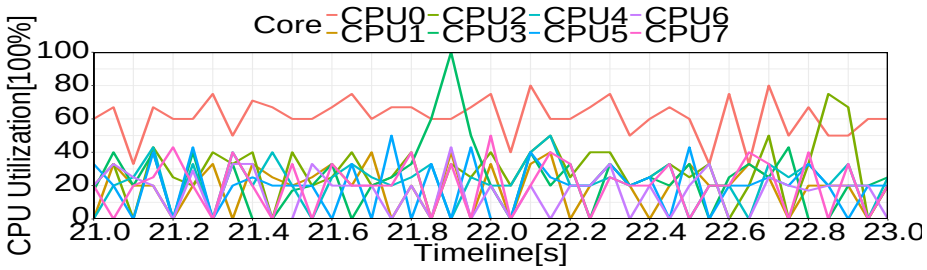
We analyze the blktrace to verify the effectiveness of optimistic stable page at the block layer. We observe more traces like the one shown in table 6 when dirty pages are being flushed out. This suggests that HTTPD processes are not blocked when they are modifying the in-flight dirty pages. We also see $(I \rightarrow D)$ and $(D \rightarrow C)$ values are close to the ones of normal flushing dirty pages.



(a) Request queue for Apache, Tomcat and MySQL tier, the queue peaks are lower than conservative stable page.



(b) Abrupt dirty page cache size drops suggest Linux is flushing dirty pages.



(c) One core is saturated while the remaining core are not idle, suggesting processes are not blocked by flushing dirty pages.

Figure 15: The effectiveness of optimistic stable page: it can lower the queue length peaks when Linux is flushing dirty pages.

4.7 Conclusion

We demonstrate the limitations of CFQ: (1) It ignores CPU priorities: I/O priority is not derived from the CPU priority, which means high CPU priority task may not get more I/O resource than low CPU priority task. (2) It ignores I/O priorities: writeback process (priority-4) submits all the writes on behalf of the process, no matter what I/O priority it is. Stable page complicates I/O priorities, it creates a situation is somewhat similar to the priority inversion problem in scheduling where a high priority task cannot make any progress as a low priority task has a resource it requires.

We found a new millibottleneck caused by stable page, a mechanism doesn't allow

a process to modify a page while the writeback operation is in progress. Asynchronous write issued by writeback daemon process is supposed to have lower priority than synchronous write issued by HTTPD and Tomcat process. But conservative stable page makes synchronous write waiting for asynchronous write, causing HTTPD and Tomcat process stalled, creating VLRT requests. We applied a remedy, optimistic stable page to eliminate the millibottlenecks. The core idea is that it checks if the given page is related to a backing device that requires page contents to be held stable during writeback. If so, then it will wait for any pending writeback to complete. Given that our hardware does not require data integrity, experimental results show our remedy significantly improve average response time by 4X.

CHAPTER V

STUDY CASE 2: LIMITATIONS OF LOAD BALANCING MECHANISMS FOR N-TIER SYSTEMS IN THE PRESENCE OF MILLIBOTTLENECKS

5.1 Background on Current Apache Load Balancing

In this paper, we study primarily the load balancers that Apache web server uses when choosing an appropriate Tomcat application server. Apache tomcat connector (mod_jk) [10] is the module that implements (two) load balancing policies:

1. Request-based (default policy): ranks the Tomcat servers according to their accumulated number of requests served (fewest as the best candidate).

2. Traffic-based: ranks the Tomcat servers according to the accumulated messages exchanged between Apache and Tomcat (fewest as the best candidate).

These policies are implemented by a two-level scheduler. The higher level (one for each policy) calculates the ranking of each server, according to each load balancing policy. For example, the number of requests (policy 1) and number of messages (policy 2) are translated into a normalized value, called lb_value. The lower level (same for all policies) uses the lb_value to schedule the next request.

5.2 Millibottlenecks Interfere With Load Balancers

For completeness, we include a short introduction to millibottlenecks here. Readers who are familiar with millibottlenecks, previously called very short bottlenecks [53] and transient bottlenecks [51], can skip to the next section.

Past studies have examined and delivered valuable insights of causes on the VLRT requests. VLRT requests can have very different causes, traversing the system stack. These include CPU dynamic voltage and frequency scaling (DVFS) control at the architecture layer, Java garbage collection (GC) at the system software layer, virtual machine (VM)

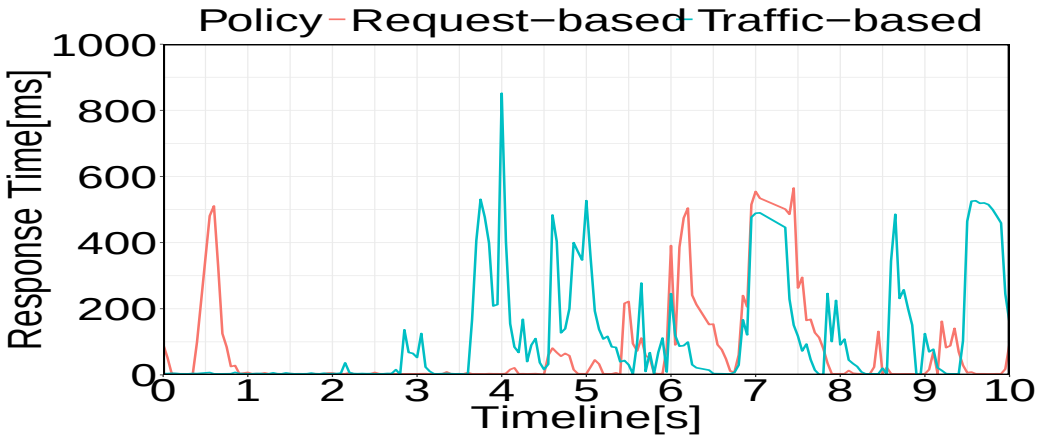


Figure 16: Point-in-time response time of the request-based and traffic-based policies during the first 10 seconds of the experiment.

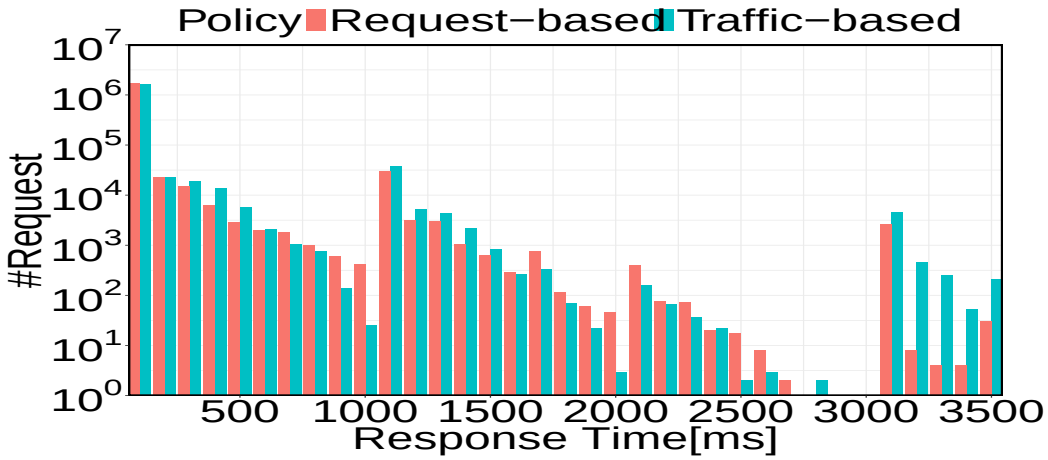


Figure 17: Frequency of requests by their response times under the request-based and traffic-based policies.

consolidation at the VM layer and bursty workloads [51–53]. The millibottlenecks in our experiments are caused by flushing dirty pages.

Our experimental results demonstrate the load balancing instability. The load balancer sends all of the requests to the candidate already suffering from millibottlenecks, which amplifies the magnitude of VLRT request. The load balancing instability was not revealed in the absence of millibottlenecks.

We ran the RUBBoS benchmark under two load balancing policies (request-based and traffic-based policy) at 70000 clients. Our experiments use 4 Apache servers, 4 Tomcat servers, and 1 MySQL server. When looking at statistical average metrics such as response

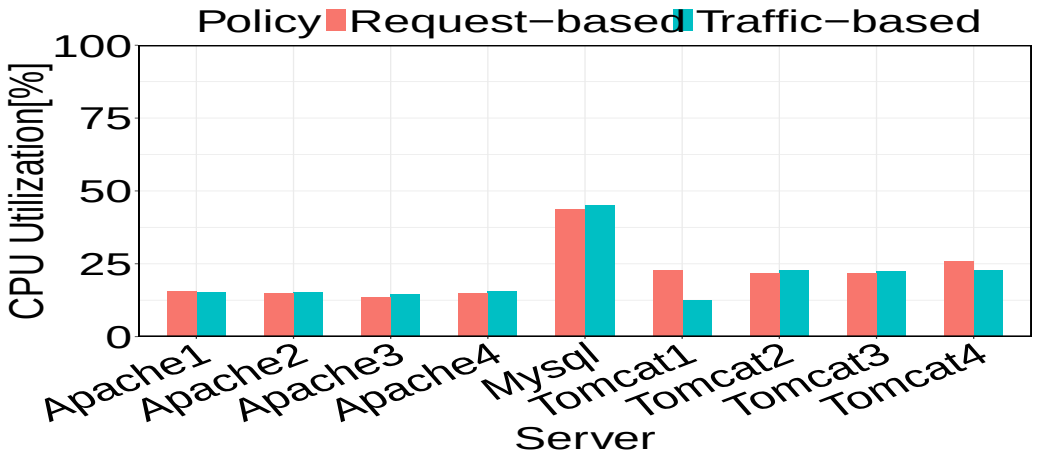
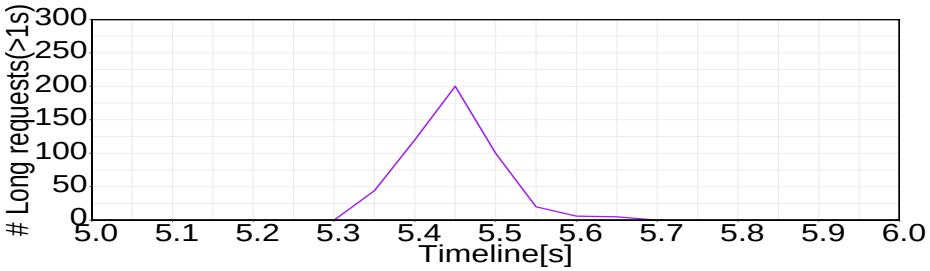


Figure 18: Average CPU usage among component servers under the request-based and traffic-based policies.

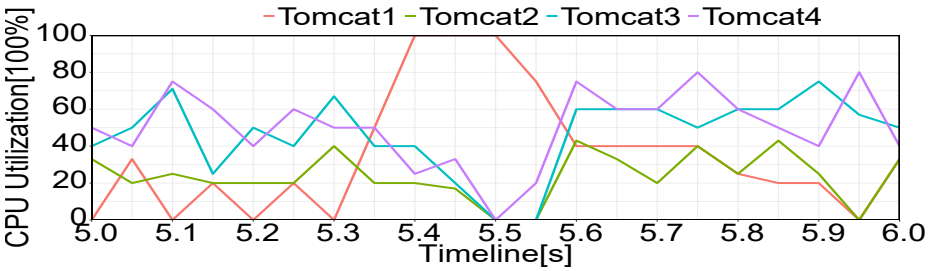
time, the performance under request-based and traffic-based policy are acceptable: their average response times are below 100ms. However, response time under the two policies present large fluctuations, as shown in Figure 16. The average system response times are not representative of the actual system performance.

The latency long-tail problem happens when very long response time (VLRT) requests arise. Under the request-based policy, 89 percent of requests finish in 10ms, VLRT (>1000ms) requests account for 5 percent of total requests. This means VRLT requests are responsible for the tremendous increase in average response time. Traffic-based policy has a similar latency long-tail problem. By plotting the distribution of request response time, latency long-tail problem can be seen more clearly. As shown in Figure 17, we observe 3 clusters of VLRT requests (at 1s, 2s and 3s). Our result is consistent with previous research, which shows VLRT requests occur even when all system components are far from saturation. As shown in Figure 18, all the component servers were at moderately low CPU utilization (the highest average CPU usage among the servers is 45%).

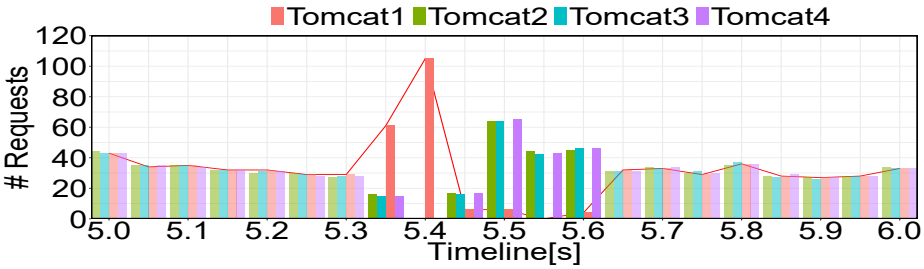
Interestingly (and somewhat surprisingly), millibottlenecks in the Tomcat tier cause more severe performance degradation compared to the case without the load balancer. As shown in Figure 19(a), the request-based policy introduced more VLRT requests compared to the experiment without the load balancer. To eliminate the interference from milibottlenecks in Apache, we increased the memory that holds the dirty pages to 4.8 GB and



(a) Number of VLRT (>1000ms) requests counted at every 50ms time window.



(b) Transient CPU saturation of Tomcat1 correlates with its queue peak, confirming Tomcat1 has a millibottleneck.



(c) Apache1 workload distribution confirms the load balancing instability: requests were sent to the candidate suffering from the millibottleneck.

Figure 19: VLRT requests are caused by millibottlenecks, and amplified by the request-based policy instability

lengthened the flushing interval to 600 seconds to ensure no millibottleneck happens on Apache during the experiment runtime. The VLRT requests are initially caused by millibottlenecks. We zoom into a period during which Tomcat experiences a millibottleneck and analyze the server-specific queues to identify the specific server experiencing the millibottleneck. We then link the VLRT requests to millibottlenecks with a fine-grained CPU utilization plot of each Tomcat server, as shown in Figure 19(b). A careful comparison of Figure 19(b) and Figure 19(a) reveals a very short period when Tomcat1 reaches full (100%) utilization coinciding with the VLRT requests. Due to space constraints, we exclude the details of diagnosing the millibottleneck. The cause of the millibottleneck is the same as

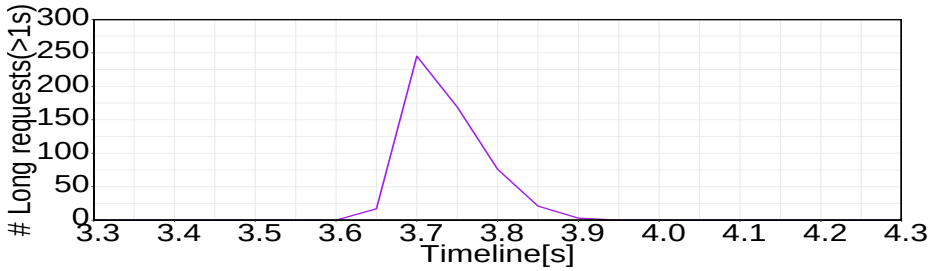
the one we discussed in section 4.2 flushing of dirty pages on the Tomcat servers.

VLRT requests are amplified by the load balancer instability: requests are sent to the Tomcat suffering from millibottlenecks instead of available healthy Tomcats. We zoom into a period in which only Tomcat1 has a millibottleneck to see how the request-based policy deals with the millibottleneck. Recall that request-based policy will send all the requests to the Tomcat with the millibottleneck. To verify the instability, we plot one Apache server's workload distribution during the period in which Tomcat 1 has a millibottleneck, as in Figure 19(c). The 4 Apaches have the same workload distribution pattern:

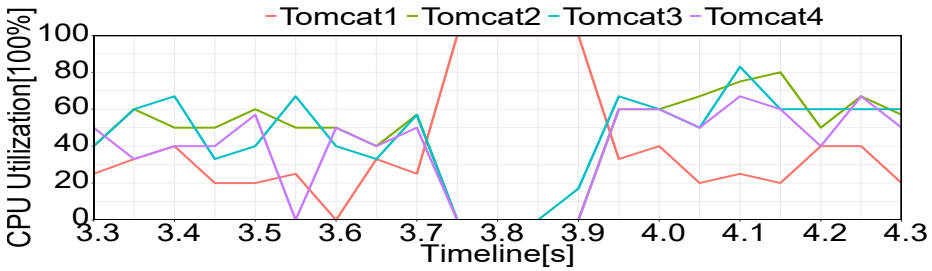
- 5.00s-5.30s (phase 1): In this period, there is no millibottleneck, and the load balancer distributes the workload evenly among the Tomcats.
- 5.35s-5.40s (phase 2): When Tomcat2, 3 and 4 are all idle, all the requests are routed to the Tomcat that has a millibottleneck (Tomcat1).
- 5.45s-5.60s (phase 3): Tomcat1 recovers from the millibottleneck, and almost all of the requests are assigned to Tomcat 2, 3 and 4, to compensate for the uneven workload distribution during the period when all the requests were sent to Tomcat1. We call this period, the recovering period.
- 5.65s-6.00s (phase 4): No millibottleneck happens, and the load balancer functions as designed.

In the period when Tomcat 1 has a millibottleneck, all requests are sent to it, causing the requests to queue in Tomcat 1. If the requests were assigned to other Tomcats without millibottlenecks, we would expect the queue length peak of Tomcat 1 to be much lower. We would also expect the queue length peaks of Apache tier would be much lower when considering the effect of queue amplification, as mentioned in Section 4.2. So the millibottleneck alone didn't create all the VLRT requests, the scheduling issue amplified the magnitude of the VLRT requests.

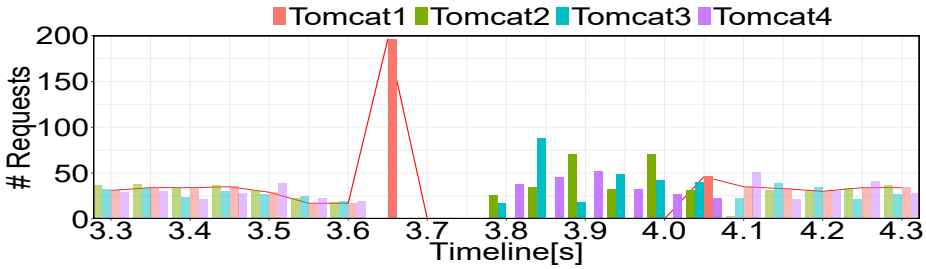
The traffic-based policy has the same instability as the request-based policy. To capture the scheduling issue visually, we plot the workload distribution of 4 Apaches during a period in which one Tomcat has the millibottleneck, as shown in Figure 20(c). By comparing it



(a) Number of VLRT (>1000ms) requests counted at every 50ms time window.



(b) Transient CPU saturation of Tomcat1 correlates with its queue peak, confirming Tomcat1 has a millibottleneck.



(c) Apache1 workload distribution visually verified the load balancing instability: requests were sent to the candidate with the millibottleneck.

Figure 20: VLRT requests are caused by millibottlenecks, and amplified by the traffic-based policy instability.

with the queue length graph in Figure 20(a) and the CPU utilization graph in Figure 20(b), we found all of the requests get routed to the Tomcat that has a millibottleneck until the millibottleneck is resolved.

5.3 Implementation Limitations of Load Balancers and Remedies

We begin discussing of the limitations of load balancing by reviewing their implementation (mechanisms). Specifically, the Apache load balancers assume that Tomcat servers are in one of three states during normal execution (not in recovery):

1. Available: the server is able to process the request.

2. Busy: all connections to the server are in use for requests.

3. Error: the server encountered an internal error and is not reachable.

First, the load balancer chooses the available server with the lowest (best) lb_value. Second, the load balancer attempts to get the chosen server to receive the request (called an endpoint). Servers that fail to return an endpoint are moved from the Available state to the Busy state. Third, the load balancer retries (to get an endpoint from) the Busy server. If the retries fail after a specified number, the Busy server is moved to the Error state. The 3-state assumption is very reasonable under the assumption of a stable system state. However, the three states become insufficient when we take millibottlenecks into account.

As stated earlier, the millibottlenecks are very short resource bottlenecks on the order of tens to hundreds of milliseconds. The servers are effectively unavailable during periods with the millibottlenecks, but then they become available again after the millibottlenecks resolve.

Algorithm 1: Pseudo code of *get_endpoint*

```
1  while ((retry * JK_SLEEP_DEF) <aw → cache_acquire_timeout) do
2      Iterate through the thread pool, try to find connected endpoint;
3      if no connected endpoint found then
4          Iterate through the thread pool again to use the first free one;
5      end
6      if get free endpoint then
7          return true;
8      else
9          retry++;
10         sleep(JK_SLEEP_DEF);
11     end
12     return false;
13 end
```

The current implementation of Apache load balancer (pseudo code shown in Algorithm 1 keeps checking if the candidate server has a free endpoint until a timeout elapses. If the candidate server doesn't have a free endpoint, it will sleep for a very short period and check again. The default value of cache_acquire_timeout is 300ms and the default sleep time is 100ms. When the load balancer checks for a free endpoint, it does not update the candidate's

state or lb_value, while it is waiting for a free endpoint. That is, the server maintains the Available state during this waiting period. This is OK for a permanent failure, since the short waiting period does not impact the ultimate state assigned by the load balancer to the candidate. However, millibottlenecks will lead the load balancer to consider the server to be Available or Busy.

To correct this situation, we modified the source code of get_endpoint to take millibottleneck state into consideration. We adopted a conservative approach: treat millibottleneck state as busy state. The idea is very straightforward: when the load balancer tries to find a free endpoint from the candidate, if the candidate cannot respond, the load balancer should skip it and move it to busy state instead of continuing to check it for a very short period.

We used the conservative approach for two reasons: First, it is hard to distinguish millibottleneck from permanent failure [40], and the conservative approach could ensure reliability. Secondly, to handle the latency and throughput demands of large-scale web services, the load balancer should make decisions quickly.

In the first test, we ran the modified mod_jk under the same workload as previous experiments and configured the load balancing policy to request-based. To verify the effectiveness of the modified get_endpoint, we plot the queue length graph firstly in Figure 21. We see the queue lengths of Tomcat and Apache tier are much lower than original request-based policy. This is because the modified request-based policy avoids sending requests to Tomcat with the millibottleneck, the reason behind it is that the Tomcat with the millibottleneck will not become the candidate, so all request were sent to the available Tomcats.

Workload distribution graph is a more intuitive way to verify the effectiveness of modified get_endpoint. We zoom into a period during which one Tomcat had millibottleneck, as shown in Figure 22(a), a queue spike of Tomcat 2 suggests that it had the millibottleneck. Note that the number of queued requests in Tomcat 2 is 200, which is 1/4 of the original request-based policy. Because of space constraints and the 4 Apaches have the same workload distribution pattern, we only show Apache1's workload load distribution graph in Figure 22(b). We focus on the period in which Tomcat 2 had the millibottleneck[0.80s - 1.15s], we see all the requests were routed to the Tomcats without millibottleneck.

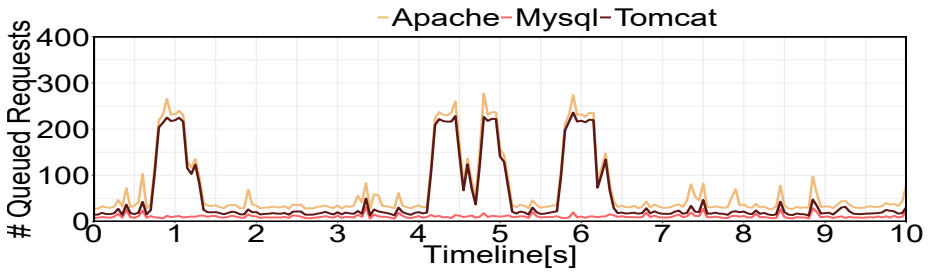
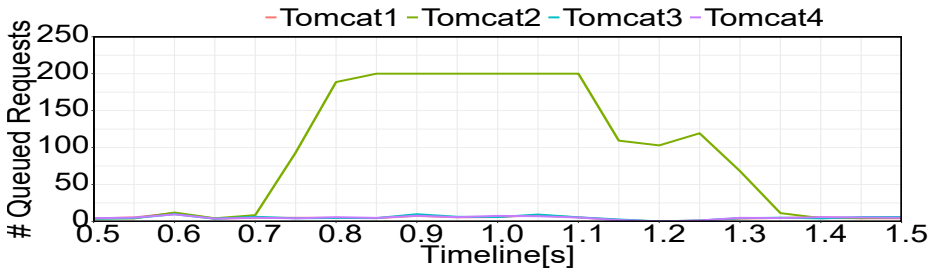
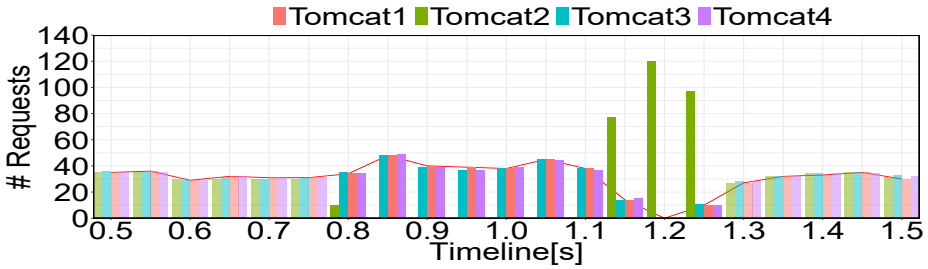


Figure 21: Queued requests in Apache, Tomcat and MySQL tier under request-based policy with modified get_endpoint. Our remedy at mechanism reduced the queued request by 75%.



(a) Queued requests in Tomcat servers, the small peak in Tomcat2's queue suggests Tomcat2 has a millibottleneck.



(b) Apache1 Workload distribution: after overcoming limitations at mechanism level, the request-based policy doesn't send requests to the candidate with millibottlenecks.

Figure 22: Remedy at mechanism level can avoid the load balancer instability.

5.4 Policy Limitations Of Load Balancers and Remedies

The first policy we study is request-based policy, which says to choose the best Tomcat server based on the fewest total number of requests. Note that lb_value here represents the number of request a candidate has served. Pseudo code for calculating the lb_value under the request-based policy is shown below:

The load balancer picks the candidate with the lowest lb_value, then it queries the candidate by calling the function get_endpoint to determine if it has a free endpoint to serve the request. If the candidate is available, Apache updates its lb_value. Traffic-based

Algorithm 2: Request-based Policy

```
1 Pick Tomcat_i;
2 if Tomcat_i has a free endpoint to serve the request then
3 |   Tomcat_i.lb_value+ = lb_mult;
4 |   Send the request to Tomcat_i;
5 |   Receive the response from Tomcat_i;
6 end
```

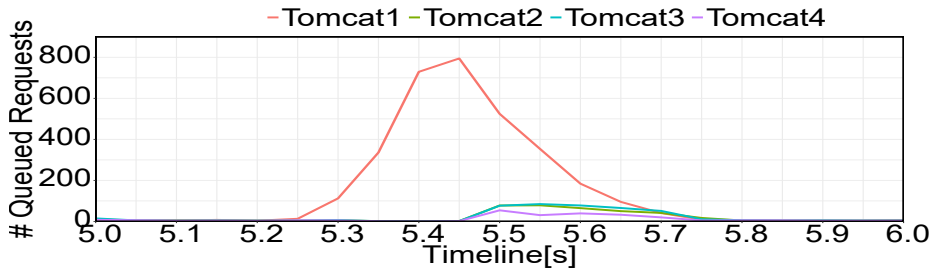
policy follows the same procedure as request-based policy, but it uses a different formula to calculate lb_value, which represents number of messages. The pseudo code for the traffic-based policy is shown below:

Algorithm 3: Traffic-based Policy

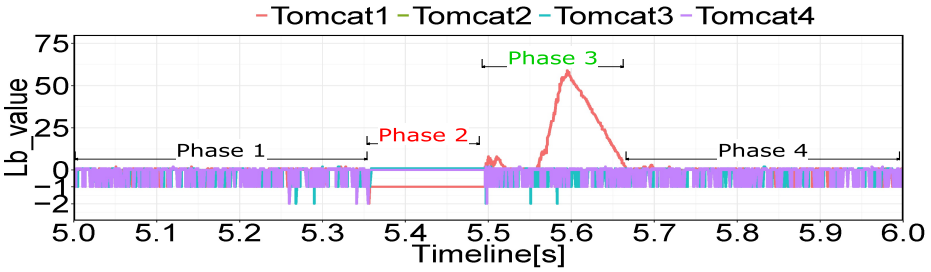
```
1 Pick Tomcat_i;
2 if Tomcat_i has a free endpoint to service the request then
3 |   Send the request to Tomcat_i;
4 |   Receive the response from Tomcat_i;
5 |   Tomcat_i.lb_value+ =(read + write sizes of the request)*lb_mult;
6 end
```

Due to the mechanism limitation, the lb_value of candidate suffering from millibottlenecks is valid. Due to the policy limitation, its lb_value is the lowest among candidates until the millibottleneck is resolved because other healthy Tomcats' lb_values keep increasing because they can process requests. Firstly, we measured the lb_value under the total request policy to confirm the instability. The request-based policy always picks the candidate with the lowest lb_value, which is the lowest number of completed requests in its semantic. We instrumented code to obtain the lb_value of each candidate. We choose the lb_value of one Tomcat without millibottleneck (Tomcat2) to draw a more clear distinction between the values. Compared to the queue length in Figure 23(a), we observe the common pattern of lb_value changes among the 4 Apaches. Because of space constraints, we only present the lb_value changes in the first Apache in Figure 23(b). The pattern is shown below:

- During the normal (without millibottlenecks) periods (phase 1 and phase 4): the lb_value of the 4 Tomcats are identical (i.e., the difference between them is at most 1) since the request-based policy evenly distributes the requests among the 4 Tomcats.



(a) Queued requests in each Tomcat server, the huge peak suggests a millibottleneck happens in Tomcat1.

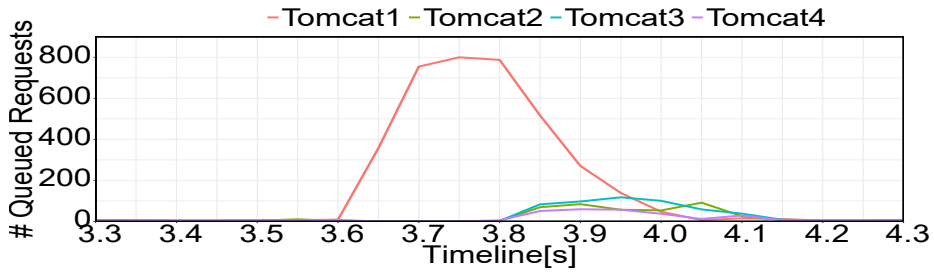


(b) Lb_values of 4 Tomcats: the candidate suffering from a millibottleneck has the lowest lb_value, causing all the requests being sent to it.

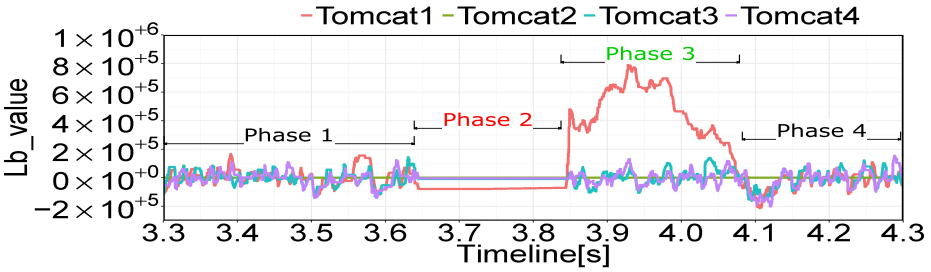
Figure 23: Policy limitations of the request-based policy lead to the load balancer instability.

- During the period in which Tomcat1 has the millibottleneck (phase 2), the lb_value of Tomcat1 is the lowest among the 4 Tomcats, which explains why it was picked. As shown in Figure 23(b), the red line is one lower than the other lines in phase 2.
- During the recovery period (phase 3), the lb_value of Tomcat1 is the highest among the 4 Tomcats, as shown by the red peak in Figure 23(b). In phase 2, each Apache accumulates tens of outgoing requests to directed Tomcat1 while no request was sent to other candidates. After Tomcat1 becomes available, it starts to process the requests that have accumulated in phase 2. The other candidates have no such accumulation, which explains why Tomcat1's lb_value increases faster than the other candidates.

The second policy we study is traffic-based policy. We verified the policy limitation of the traffic-based policy by plotting the lb_values of the four candidates. The traffic-based policy's pattern is similar to the request-based policy's: the candidate experiencing a millibottleneck has the lowest lb_value. Recall that the traffic-based policy also picks the candidate with the lowest lb_value. In this setting, lb_value is the sum of the read and



(a) Queued requests in each Tomcat server, the huge peak suggests a millibottleneck happens in Tomcat1.



(b) Lb_values of 4 Tomcats: the candidate suffering from a millibottleneck has the lowest lb_value, causing all the requests being sent to it.

Figure 24: Policy limitations of the traffic-based policy lead to the load balancer instability.

write sizes of all of the requests that a candidate has served. We choose the lb_value of one Tomcat without a millibottleneck (Tomcat2) to establish a baseline against with which to compare. We observe the common pattern among the changes in lb_value among the 4 Apaches. Because of space constraints, we only show the first one in Figure 24(b) and don't discuss the details of the pattern here.

The load balancers made the following assumptions: workload is stable and there is no 100% bottleneck on the Tomcat server. These load balancing policies making decisions based on accumulated resource utilization can work well in the absence of millibottlenecks; however, they suffer from limitations in their implementations when millibottlenecks occur. The policy limitations can make the instability even more severe. In the original request-based and traffic policies, the candidate experiencing a millibottleneck is treated as available, and the lb_value of that candidate will not be updated due to the millibottleneck. In contrast, those available candidates' lb_value will keep increasing. This leads the candidate with a millibottleneck to have the lowest lb_value, Accordingly, the load balancer proceeds to send all requests to the candidate with millibottleneck, causing all requests to queue in

that candidate.

These load balancing policies making decisions based on accumulated resource utilization are designed to behave stable under different conditions. The problem faced with these policies is that they are linear, they are not adaptive to changes. To overcome the limitation, the load balancer can acquire additional state information and attempt to make optimistic scheduling decisions based on the current state. Here we introduce a policy using current information to make scheduling decision: current_load policy, the core idea is that it picks the candidate with the least current workload. Pseudo code for current_load policy is shown below, it keeps track of how many requests each candidate is currently assigned at present.

Algorithm 4: Current_load Policy

```
1  Pick Tomcat_i;
2  if Tomcat_i has a free endpoint to service the request then
3  |   Tomcat_i.lb_value+ = lb_mult;
4  |   Send the request to Tomcat_i;
5  |   Receive the response from Tomcat_i;
6  |   if Tomcat_i.lb_value ≥ lb_mult then
7  |   |   Tomcat_i.lb_value− = lb_mult;
8  |   else
9  |   |   Tomcat_i.lb_value = 0;
10 |   end
11 end
```

Current_load policy has two advantages: First, it is more adaptive to the millibottlenecks. The candidate suffering from millibottlenecks is unlikely to process the request. Compared to available candidates, it has more requests being served from the perspective of the load balancer. Second, it is robust to the mechanism limitations. Even though Apache could be stuck in calling get_endpoint on the Tomcat with the millibottleneck, the lb_value of the candidate with the millibottleneck remains the highest among the candidates. Current_load policy doesn't rely on the 3-stable-state assumption: lb_value of the candidate with the millibottleneck will be the highest among the candidates, which means the candidate with the millibottleneck will not be picked. The implementation limitation of get_endpoint will proceed only after the candidate with millibottleneck is picked. So current_load policy can alleviate the implementation limitations at mechanism level. The

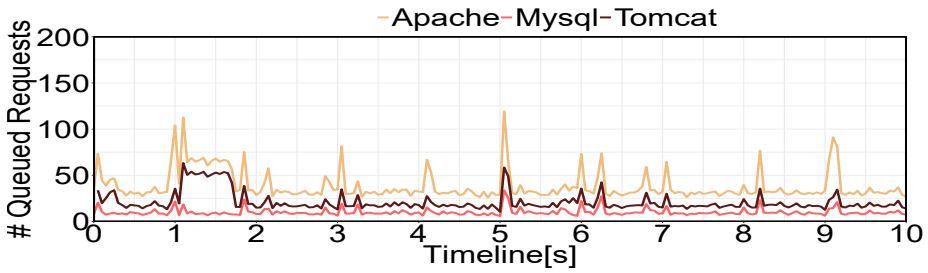


Figure 25: Queued requests in Apache, Tomcat and MySQL tier under current_load policy. The absence of huge queue peak in the presence of millibottlenecks suggests our remedy at policy level can avoid the scheduling instability.

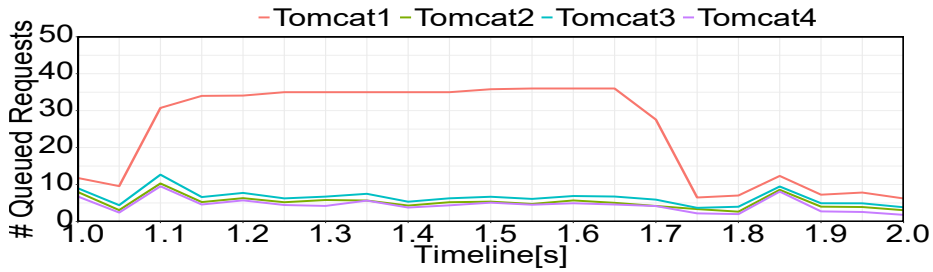
Table 7: Performance of request-based policy, traffic-based policy and our remedies at policy and mechanism levels

Policy	# Total Requests	Average response time (ms)	% VLRT requests (>1000 ms)	% Normal requests (<10 ms)
Original request-based	1791857	41.00	5.33%	88.85%
Original traffic-based	1789493	55.50	6.89%	85.55%
Current_load	1801765	3.62	0.21%	96.70%
Modified request-based	1800562	4.87	0.55%	95.82%
Modified traffic-based	1799662	5.87	0.76%	93.93%
Modified current_workload	1796697	3.60	0.20%	96.67%

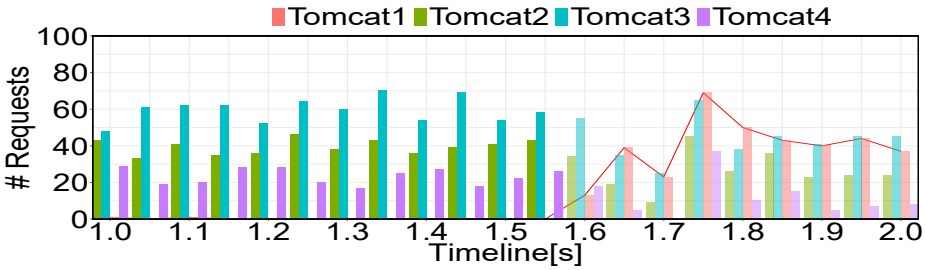
shortcoming of current_load policy is rapidly changing system state may cause the policy to react in an unstable manner.

We then use experimental data to show current_load policy is more adaptive to the millibottlenecks. We ran the same workload as request-based and traffic-based policy. First, we examine the queue length graph in Figure 25, we see there is barely any huge spike in Tomcat tier, the reason is that current_load policy is able to avoid sending requests to Tomcat with the millibottleneck. We also observe that the number of spikes of Apache tier is less than request-based and traffic-based policy, this could be explained by the disappearance of queue amplification starting from Tomcat tier. The millibottlenecks are inevitably in Tomcat tier, current_load policy can reduce the magnitude of the VLRT requests by avoiding the scheduling issue.

It is inevitable for the foreground task to experience some delay under heavy asynchronous I/Os for the following reasons. First, there can be a contention in holding a lock



(a) Queued requests in Tomcat servers, the small queue length peak suggests Tomcat1 has a millibottleneck.



(b) Apache1 Workload distribution: the current_load policy sends all requests to the available candidates instead of the candidates suffering from millibottlenecks.

Figure 26: Workload distribution further confirms the current_load policy can avoid the scheduling instability.

to modify the file system metadata. Second, it is possible that another asynchronous I/O is already in progress in the storage device when a synchronous I/O is dispatched. Third, there can be no room in memory or request queues for additional I/Os.

To further verify the effectiveness of current_load policy, we want to see how current_load policy distributes the workload among the candidates when the millibottleneck occur. We zoom into a period during which one Tomcat has a millibottleneck, as shown in Figure 26(a), a small spike of Tomcat 1 indicates it had millibottleneck. We found a small period of iowait saturation correlating to the spike, confirming that the millibottleneck is caused by flushing dirty pages. The Tomcat spike is not very obvious, less than 40 requests are queued when the millibottleneck happened, which is not comparable with huge and sharp Tomcat spike under request-based and traffic-based policy. Because of space constraints and the 4 Apaches have the same workload distribution pattern, we only show Apache1's workload load distribution graph in Figure 26. We focus on the period in which Tomcat 1 has the millibottleneck[1.00s - 1.55s], we see all the requests are routed to the Tomcats without millibottlenecks.

5.5 Summary of Comparison

VLRT requests are caused by millibottlenecks, amplified by the scheduling issue. We attribute the majority of VLRT requests to three factors: (a) number of the millibottleneck; (b) severity of the millibottleneck and (c)scheduling policy, more concretely, whether the load balancer can avoid sending requests to the Tomcat with the millibottleneck. Since experiment duration, flushing dirty page interval and the total size of dirty pages are almost fixed, we think factor (a) and (b) are fixed across the tests. Such that factor (c) determines the performance. Even millibottlenecks are inevitable, load balancers are supposed to avoid sending requests to the Tomcat with millibottleneck, minimizing the effect of millibottleneck on performance. The combination of limitations at policy and levels will lead to the scheduling instability. Overcoming limitations at least at one level guarantee requests will not be sent to the candidate with millibottlenecks.

To measure the effect of scheduling instability on performance, we first conduct a comparison of the average response time, as shown in Table 7. Our remedy at policy level, current_load policy, can improve average response time by 12x and 15x compared to request-based policy and traffic-based policy respectively. Current_load policy can avoid the scheduling instability even though the implementation limitations still exist at mechanism level. Our remedy at mechanism level, modified get_endpoint function, can achieve almost the same performance improvement.

To further verify the effectiveness of our remedies to reduce the VLRT requests amplified by the load balancer instability, we investigate the percentage of VLRT requests (>1000ms), as shown in Table 7. We found that current_load policy and fixed get_endpoint reduce the number of the VLRT requests significantly. The percentages of VLRT requests under the request-based policy and traffic-based policy were 5.3% and 6.9%, respectively. Current_load policy, request-based policy with modified get_endpoint and traffic-based policy with modified get_endpoint reduces the percentage of VLRT requests to 0.2%, 0.55% and 0.76% respectively. These VLRT requests are only caused by millibottlenecks. While under request-based and traffic-based policy, the VLRT requests are caused by millibottlenecks and amplified by scheduling issue. We can draw a conclusion based on the comparison:

scheduling instability is the dominating factor that leads to the VLRT requests (more that 96% of VLRT requests are caused by scheduling issue). The current_load policy with modified get_endpoint is the case where we overcome limitations at both policy and mechanism levels, but will not gain further improvement because they achieve the same goal.

5.6 Conclusion

In this section, we have identified several limitations of some load balancing policies and mechanisms in the presence of millibottlenecks. Our experimental results demonstrate the load balancers instability, where new requests are sent to the candidate suffering from millibottlenecks instead of idle ones, amplifying the magnitude of very long response time request caused by the millibottlenecks. The instability is caused by limitations at both mechanisms and policy levels. At mechanism level, millibottlenecks are often mistakenly treated as available. To overcome the limitation, we can treat millibottlenecks as busy state. At policy level, policies making decisions based on accumulated resource utilization cannot react to millibottlenecks. Our remedy is using policies that make scheduling decision based on candidate's current state. We have shown the two remedies can improve the RUBBoS application's response times by a factor of 12 when facing the millibottlenecks caused by flushing dirty pages.

CHAPTER VI

RELATED WORK

6.1 Performance Analysis

There are efforts focusing on exploring sources of poor tail latency in high large scale distributed applications [19,20,29,41,42,48,51–54,57]. Dean et al. [29] present their approaches to bypass/mitigate tail latency in Google's large-scale software application. PriorityMeister [57] automatically and proactively configures workload priorities and rate limits across multiple stages to meet tail latency SLOs for shared networked storage. Suresh et al. [48] propose an adaptive replica selection mechanism to reduce tail latencies, the core idea is using a combination of in-band feedback from servers to rank and prefer faster replicas along with distributed rate control. Bobtail [54] proactively detects and avoids these bad neighboring VMs to avoid long tail problem caused by co-scheduling of CPU-bound and latency-sensitive tasks.

A plethora of approaches are proposed to diagnose performance problems. Li et al. [42] attempt to identify the hardware, operating system, and application-level sources of poor tail latency in high throughput servers executing on multi-core machines. Wang et al. [52] propose a statistical correlation analysis between a server's fine-grained throughput and concurrent jobs in the server to infer the server's real-time performance state. Cohen et al. [28] use a class of probabilistic models (Tree-Augmented Bayesian Networks) to identify combinations of system-level metrics and threshold values that correlate with SLO violations. Chow et al. [27] analyze end-to-end performance of large-scale Internet services through 3 steps: [1] generating a causal model of system behavior via reasoning over software component logs, [2] generating potential hypotheses about program behavior, [3] rejecting hypotheses contradicted by the empirical observations. Aguilera et al. [18] propose approaches to infer the dominant causal paths trough a distributed system from the traces without modifying the system or having semantic knowledge about it. Koskinen et al. [37] obtain precise

traces for black-box system without application-specific instrumentation, however, it relies on knowledge of protocols to isolate events or requests. Chopstix [21] collects profiles of low-level OS events at the granularity of executables, procedures and instruction. Then these events are reconstructed to diagnose problems in a large-scale production system.

6.2 IO priority

More recently, research has been looked into priority issue in I/O stack. Jeong et al. [35] introduce a new type of I/O operations called Quasi-Asynchronous I/O (QASIO). QASIOs are defined as the I/O operations which are issued asynchronously, but should be treated as synchronous I/Os since other tasks are blocked on them. They also propose a novel scheme to detect QASIOs at run time and boost them over the other asynchronous I/Os in the I/O scheduler. Yang et al. [35] demonstrate that traditional block-level I/O schedulers are unable to meet throughput, latency, and isolation goals. They implemented a split-level I/O scheduling, a new framework that splits I/O scheduling logic across handlers at three layers of the storage stack: block, system call, and page cache.

There are a few solutions to the priority inversion problem in real time systems. One solution is to turn off all system interrupts, effectively halting thread preemption in the system, while critical tasks execute. However, to make this work, we cannot implement more than two thread priorities, and critical sections where resources are locked need to be very brief and tightly controlled. Another solution is priority ceiling protocol [46], The goal of this protocol is to prevent the formation of deadlocks and of chained blocking. This protocol ensures that when a job A preempts the critical section of another job B and executes its own critical section z, the priority at which this new critical section z will execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections. The problem can be addressed in priority-based scheduling algorithms by supporting priority inheritance protocol [46]. The basic idea of priority inheritance protocols is that when a job blocks one or more higher priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks. After exiting its critical section, the job returns to its original priority level.

6.3 Scheduling

There is a long history of work in academia and industry to study load balancer [25, 26, 31, 32, 39, 43, 47, 49, 56]. Cardellini et al. [25] proposed a classification of existing approaches based on the entity that dispatches the client requests among the distributed servers: client-based, DNS-based, dispatcher-based, and server-based. They also analyzed the efficiency and limitations of the various approaches and their trade off. By using simple analytic models, eager et al. [32] demonstrate that extremely simple adaptive load sharing policies, which collect very small amounts of system state information and which use this information in very simple ways, dramatically better than when no load sharing is performed, and nearly as well as more complex policies that utilize more information. Teo et al. [49] conduct an experimental analysis of the performance and scalability of scheduling algorithms including round robin, least connected and least loaded. However, these previous works assume the servers are in stable state: either available or failed. Their goal is to achieve good response time and resource utilization. In contrast to these works, our work deals with avoiding long tails in response times in the presence of millibottlenecks. To the best of our knowledge, our paper is the first to systematically analyze load balancer performance in the presence of millibottlenecks.

Recent research adds latency as a metric to evaluate the performance of scheduler. To achieve low latency, Sparrow [44] uses the similar conservative mechanism as we propose when scheduler failure happens: if the scheduler has failed, the client connects to the next scheduler in the list. Sparrow also triggers a callback at the application, which allows the frameworks to decide how to handle tasks that were in-flight during the scheduler failure. To avoid scheduling tasks to repeated offenders, Boutin et al. [22] developed a server failure model for the scheduling framework by mining historical data and various factors to predict the likelihood of possible repeated server failures in the near future. Corona [11] uses push-based scheduling instead of pull-based. After the cluster manager receives resource requests from the job tracker, it pushes the resource grants back to the job tracker. Also, once the job tracker gets resource grants, it creates tasks and then pushes these tasks to the task trackers for running. Scheduling latency can be minimized due to there is no periodic

heartbeat involved in this scheduling.

CHAPTER VII

CONCLUSION AND FUTURE WORK

To summarize, in this dissertation we propose a novel approach to detect system level millibottlenecks by fine-grained monitoring of locks. We conduct an illustrative scenario in which our approach successfully diagnoses the root cause of millibottleneck caused by conservative stable page, where the file system operations are blocked until one or more asynchronous I/O operations are completed. This situation is somewhat similar to the priority inversion problem in scheduling where a high priority task cannot make any progress as a low priority task has a resource it requires. The system level millibottleneck we have found is only a humble beginning. The fundamental cause of this class of millibottlenecks is that CFQ scheduler ignores the priorities of asynchronous writes, we envision a new class of system level millibottlenecks caused by priority inversion exists in Linux kernel, especially in I/O stack. Nevertheless, we believe that the search the class of millibottlenecks and the remedies will be an interesting journey.

Load balancers in N-tier systems have proven to work well in stable environments. However, we have identified several limitations of some load balancing policies and mechanisms in the presence of millibottlenecks. Our experimental results demonstrate the load balancers instability, where new requests are sent to the candidate suffering from millibottlenecks instead of idle ones, amplifying the magnitude of very long response time request caused by the millibottlenecks. The load balancer instability is caused by limitations at both mechanisms and policy levels. At mechanism level, millibottlenecks are often mistakenly treated as available. To overcome the limitation, we can treat millibottlenecks as busy state. At policy level, policies making decisions based on accumulated resource utilization cannot react to millibottlenecks. Our remedy is using policies that make scheduling decision based on candidate's current state. We have shown the two remedies can improve the RUBBoS application's response times by a factor of 12 when facing the millibottlenecks

caused by flushing dirty pages. The RUBBoS results also demonstrate the effectiveness of our remedies in reducing VLRT requests. Millibottlenecks are difficult to detect, and it is infeasible to eliminate all of them. Other load balancers in N-tier systems can take advantage of our remedies to shorten the latency tail caused by scheduling instability when facing millibottlenecks caused by other resource shortage. In general, when millibottlenecks happen, we advocate adaptive policies that take into account the current state of servers, with appropriate mechanisms that treat millibottlenecks as unavailable.

7.1 Future Work

Due to the potential depth of the proposed research, the proposed dissertation—even though self-contained and highly significant—can merely be regarded as an initial step towards one of the important goals in load balancer avoid the tail latency in the presence of millibottlenecks. One topic of particular interest is how the load balancer can incorporate machine learning capabilities in intelligent scheduling. The milliBottlenecks in n-tier systems might exhibit the similar pattern. We want to develop a millibottleneck-aware load balancer with a built-in inductive learning module, is developed for heuristic acquisition and refinement. This method enables the scheduler to classify distinct millibottlenecks patterns and to generate a decision tree consisting of heuristic policies for avoiding sending requests to the machines with millibottlenecks.

We plan to release the tool of as well as related scripts of the experiments soon. Our goal is to facilitate the experimental research on system level millibottlenecks. The performance unpredictability associated millibottlenecks does negatively impact latency overall, but little is known about the long-tail behavior. By open-sourcing our tools, more researchers could utilize our fine-grained monitoring tools on other scenarios to validate if there are more millibottlenecks in Linux kernel. To make the tool more usable, we plan to deploy a web server to provide a user interface on top of the profile database. This makes it easy to access lock statistic data and construct ad hoc queries for the use of the lock statistic data. Two visual interfaces retrieve information from the database, and both of them are navigable from a web browser. One visual interface displays the lock statistic(max hold time, max

wait time and total). This visual interface will allow users refine the query to more specific data. For example, the user can restrict the query to only report the most contended lock for a specified time window. Additionally, the user can modify or refine any of the parameters to the current query to create a custom lock statistic view. Another visual interface displays the Linux kernel dependency graph. Each vertex represents the function name, causal dependencies are represented by edges. Users can write queries against the graph to diagnose the root causes the millibottlenecks. For example, given a function v, users can issue a query to find all functions (vertices) that vertex v is causally dependent on. Queries use slicing operators, from a starting vertex v, forward slicing finds all vertices that causally and transitively depend on v, while backward slicing finds all vertices that v is dependent on. Our system level millibottleneck detection tool will become part of the milliMonitor, researchers are able to contribute more milliMonitor components, port them to other platforms, and achieve the growth of the milliMonitor ecosystem.

REFERENCES

[1] "Block io priorities." https://www.kernel.org/doc/Documentation/block/ioprio.txt.

[2] "Cfq (complete fairness queueing)." https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[3] "Cfs scheduler." https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[4] "Emulab - network emulation testbed." https://www.emulab.net//.

[5] "Linux kernel profiling with perf." https://perf.wiki.kernel.org/index.php/Tutorial/.

[6] "Optimizing stable pages." https://lwn.net/Articles/528031/.

[7] "Priority inversion." https://en.wikipedia.org/wiki/Priority_inversion.

[8] "Rubbos: Bulletin board benchmark." http://jmob.ow2.org/rubbos.html/.

[9] "Stable pages." https://lwn.net/Articles/442355/.

[10] "Tomcat connectors (mod_jk)." https://tomcat.apache.org/download-connectors.cgi/.

[11] "Under the hood: Scheduling mapreduce jobs more efficiently with corona." https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920/.

[12] "Collectl." "http://collectl.sourceforge.net/".

[13] "Egypt." "http://www.gson.org/egypt/egypt.html".

[14] "Inode locking rules." "http://elixir.free-electrons.com/linux/latest/source/fs/inode.c".

[15] "Iotop." "http://guichaz.free.fr/iotop/".

[16] "Lockstat." "http://lxr.free-electrons.com/source/Documentation/lockstat.txt?v=3.4".

[17] "sar - collect, report, or save system activity information.." "https://linux.die.net/man/1/sar".

[18] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A., "Performance debugging for distributed systems of black boxes," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pp. 74–89, 2003.

[19] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M., "Data center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74, 2010.

[20] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M., "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pp. 253–266, 2012.

[21] Bhatia, S., Kumar, A., Fiuczynski, M. E., and Peterson, L., "Lightweight, high-resolution monitoring for troubleshooting production systems," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 103–116, 2008.

[22] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L., "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 285–300, USENIX Association, 2014.

[23] Brunelle., A. D., "Block i/o layer tracing: blktrace," tech. rep., 2006.

[24] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H., "Dynamic instrumentation of production systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2004.

[25] Cardellini, V., Colajanni, M., and Yu, P. S., "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, vol. 3, pp. 28–39, May 1999.

[26] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S., "Scalable Distributed Stream Processing," in *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, (Asilomar, CA), January 2003.

[27] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F., "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 217–231, USENIX Association, 2014.

[28] Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., and Symons, J., "Correlating instrumentation data to system states: A building block for automated diagnosis and control.," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 231–244, 2004.

[29] Dean, J. and Barroso, L. A., "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[30] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[31] Eager, D. L., Lazowska, E. D., and Zahorjan, J., "A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract)," in *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '85, (New York, NY, USA), pp. 1–3, ACM, 1985.

[32] Eager, D. L., Lazowska, E. D., and Zahorjan, J., "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Softw. Eng.*, vol. 12, pp. 662–675, May 1986.

[33] Feiner, P., Brown, A. D., and Goel, A., "Comprehensive kernel instrumentation via dynamic binary translation," *SIGPLAN Not.*, vol. 47, pp. 135–146, Mar. 2012.

[34] Greg Linden, "Make Data Useful." "http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf".

[35] Jeong, D., Lee, Y., and Kim, J.-S., "Boosting quasi-asynchronous i/o for better responsiveness in mobile devices," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, (Santa Clara, CA), pp. 191–202, USENIX Association, Feb. 2015.

[36] Kohavi, R., Henne, R. M., and Sommerfield, D., "Practical guide to controlled experiments on the web: Listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, (New York, NY, USA), pp. 959–967, ACM, 2007.

[37] Koskinen, E. and Jannotti, J., "Borderpatrol: Isolating events for black-box tracing," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pp. 191–203, 2008.

[38] Lai, C. A., Kimball, J., Zhu, T., Wang, Q., and Pu, C., "milliscope: A fine-grained monitoring framework for performance debugging of n-tier web services," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 92–102, June 2017.

[39] Leland, W. and Ott, T. J., "Load-balancing heuristics and process behavior," in *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation*, SIGMETRICS '86/PERFORMANCE '86, (New York, NY, USA), pp. 54–69, ACM, 1986.

[40] Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., and Walfish, M., "Detecting failures in distributed systems with the falcon spy network," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 279–294, ACM, 2011.

[41] Leverich, J. and Kozyrakis, C., "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pp. 4:1–4:14, 2014.

[42] Li, J., Sharma, N. K., Ports, D. R., and Gribble, S. D., "Tales of the tail: Hardware, os, and application-level sources of tail latency," Tech. Rep. UW-CSE14-04-01, Department of Computer Science & Engineering, University of Washington, April 2014.

[43] Mitzenmacher, M., "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 1094–1104, Oct. 2001.

[44] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I., "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 69–84, ACM, 2013.

[45] Pu, C., Kimball, J., Lai, C.-A., Zhu, T., Jack, L., Park, J., Wang, Q., Jayasinghe, D., Xiong, P., Malkowski, S., Wu, Q., Jung, G., Koh, Y., and Swint, G., "The millibottleneck theory of performance bugs, and its experimental verification," in *Proceedings of the 37rd IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*, pp. 1–1, 2017.

[46] Sha, L., Rajkumar, R., and Lehoczky, J. P., "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, pp. 1175–1185, Sept. 1990.

[47] Shivaratri, N. G., Krueger, P., and Singhal, M., "Load distributing for locally distributed systems," *Computer*, vol. 25, pp. 33–44, Dec. 1992.

[48] Suresh, L., Canini, M., Schmid, S., and Feldmann, A., "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, (Berkeley, CA, USA), pp. 513–527, USENIX Association, 2015.

[49] Teo, Y. M. and Ayani, R., "Comparison of load balancing strategies on cluster-based web servers," *Simulation*, vol. 77, no. 5-6, pp. 185–195, 2001.

[50] Wang, L., Wang, Z., Yang, C., Zhang, L., and Ye, Q., "Linux kernels as complex networks: A novel method to study evolution," in *2009 IEEE International Conference on Software Maintenance*, pp. 41–50, Sept 2009.

[51] Wang, Q., Kanemasa, Y., Li, Jack Lai, C.-A., Matsubara, M., and Pu, C., "Impact of dvfs on n-tier application performance," in *Proceedings of ACM Conference on Timely Results in Operating Systems (TRIOS 2013)*, pp. 33–42, 2013.

[52] Wang, Q., Kanemasa, Y., Li, J., Jayasinghe, D., Shimizu, T., Matsubara, M., Kawaba, M., and Pu, C., "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS 2013)*, pp. 31–40, 2013.

[53] Wang, Q., Kanemasa, Y., Li, J., Lai, C.-A., Cho, C.-A., Nomura, Y., and Pu, C., "Lightning in the cloud: A study of transient bottlenecks on n-tier web application performance," in *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, (Broomfield, CO), USENIX Association, Oct. 2014.

[54] Xu, Y., Musgrave, Z., Noble, B., and Bailey, M., "Bobtail: Avoiding long tails in the cloud," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pp. 329–342, 2013.

[55] Yang, S., Harter, T., Agrawal, N., Kowsalya, S. S., Krishnamurthy, A., Al-Kiswany, S., Kaushik, R. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Split-level i/o scheduling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 474–489, ACM, 2015.

[56] Zhou, S., Zheng, X., Wang, J., and Delisle, P., "Utopia: A load sharing facility for large, heterogeneous distributed computer systems," *Softw. Pract. Exper.*, vol. 23, pp. 1305–1336, Dec. 1993.

[57] Zhu, T., Tumanov, A., Kozuch, M. A., Harchol-Balter, M., and Ganger, G. R., "Prioritymeister: Tail latency qos for shared networked storage," in *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, (New York, NY, USA), pp. 29:1–29:14, ACM, 2014.